Formal Verification of TCP and T/TCP

by

Mark Anthony Shawn Smith

B.S. Computer and Information Science
Brooklyn College, 1989

S.M. Electrical Engineering and Computer Science
M.I.T, 1993

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1997

Author ....
Department of Electrical Engineering and Computer Science
September 5, 1997

Certified by ....
Nancy A. Lynch
Professor
Thesis Supervisor

Accepted by ....
Author C. Smith
Chairman, Departmental Committee on Graduate Students

Formal Verification of TCP and T/TCP

by

Mark Anthony Shawn Smith

Submitted to the Department of Electrical Engineering and Computer Science
on September 5, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

In this thesis we present a formal abstract specification for TCP/IP transport level protocols and formally verify that TCP satisfies this specification. We first verify a formal model of TCP where we assume it has unbounded counters. With bounded counters, TCP requires several timing mechanisms to function correctly. We also model TCP with these timing mechanisms and verify that it also satisfies our specification. We also present a formal description of an experimental protocol called T/TCP which is designed to provide the same service as TCP, but with optimizations to make it efficient for transactions. Even with unbounded counters this protocol does not provide the same service as TCP as it may deliver the same message twice. Even though the service provide by T/TCP is not exactly the same as TCP, its behavior may be acceptable for some applications. Therefore, we define a weaker specification that captures this behavior of T/TCP while maintaining the other correctness properties of our initial specification. We then verify that T/TCP satisfies this weaker specification.

Our specifications are presented using an untimed automaton model, and we present the protocols using a timed automaton model. The formal verification is done using *invariant assertion* and *simulation* techniques.

Because of our observation that in certain situations T/TCP does not satisfy our stronger specification, we examine the question of whether any protocol can deliver streams of data reliablely and still have fast transactions. In this thesis we present a formal definition of what it means to provide both services, and then prove that it is impossible for any protocol to provide both services if the processes do not have "accurate" clocks. The formal model used to describe the system and present the proof is a novel combination of a model with liveness properties and a model that allows local clocks.

Thesis Supervisor: Nancy A. Lynch
Title: Professor

Acknowledgments

My thesis supervisor Professor Nancy Lynch has taught me a great deal about formal methods, writing proofs, and rigorous scholarship. In writing this thesis, there were many instances when Nancy through her careful reading and rigorous analysis, found errors or ambiguities in the work. She has taught me that one can never be too rigorous when dealing with complicated problems. For these lessons and for her support during my years in the theory of distributed systems group, I am truly thankful.

I would also like to thank Dr. Dave Clark for being a reader on the thesis. It was Dave who suggested the T/TCP protocol as a candidate for using formal methods to verify, and Dave is also the person that made sure that the abstract models in the thesis are reasonable representations of the actual protocols.

My other reader, Professor Albert Meyer, is also my former academic advisor, and he has been very supportive of me through my years at M.I.T.. I thank him for his advice and support through my graduate career.

My years at M.I.T. were enriched intellectually and socially through the many friendships I have made with other students there. In particular I would like to thank my Mojdeh Mohtashemi and Charles Isbell. Mojdeh has been an officemate, a roommate, a confidante, and a kindred spirit. We have shared many good times over coffee at Au bon Pain. Charles has been my workout partner, fellow hip-hop enthusiast and co-founder of CCS. I could always count on Charles to be there whenever I needed some help.

Finally, I would thank my family whose love and support has made all my endeavors possible. I especially want to thank my wife Sharon, whose love, support, and patience, has been a source of strength and comfort for me over the pass three years.

This thesis is dedicated to the memory of Anna Pogosyants, a former colleague in the theory of distributed systems group.

Contents

Chapter 1

Introduction

The original motivation for this work was to do a formal verification of an experimental transport level protocol called T/TCP. This protocol, by Braden and Clark [8, 6, 7], is designed to be a *unified transport protocol* in that it should work well for both transactions and streaming. A transaction is typically a request from a client and a response from a server. Streaming, on the other hand, is the sending of significant amounts of data. The idea behind the design of T/TCP is to extend the Transmission Control Protocol (TCP) to make it efficient for transactions (hence the name T/TCP).

1.1 Transport level protocols

TCP is the most commonly used transport level protocol on the Internet. The basic service that it provides is reliable *end-to-end* delivery of data between application programs. On the Internet packets sent from one user to another may get duplicated, lost, or arrive out of order. TCP ensures that these packets are delivered to the application programs without duplication, without loss, and in the correct order. While TCP works well for data streaming, it does not work well for transactions because it has an open phase (the three-way handshake protocol) that forces two round trips across the network for a client to send a request and get a response from a server. Ideally we would like the request and response to be done in one round trip across the network. T/TCP changes the open phase of TCP, so that in most circumstances a three-way handshake protocol is no longer required, instead a two-way handshake protocol is used. The two-way handshake protocol allows a transaction to be completed in one round trip across the network.

1.2 The correctness of T/TCP

The designers of T/TCP believed their protocol was correct since it is based on TCP, but the changes they made were sufficiently complex to make them uncertain. Therefore, they thought a formal correctness proof would be useful [8]. Our initial plan of attack for verifying T/TCP was to assume the correctness of TCP and leverage off this correctness in the verification of T/TCP. However, we could not find any work that verified TCP in sufficient generality to use in our work. Other works have verified parts of TCP or protocols similar to TCP. In [18, 35], Lampson, Lynch, and Søgaard-Andersen formally verify the correctness the five packet handshake protocol of Belsnes [4] which forms the basis of the open and close phase of TCP and ISO-TP4. However, this work does not verify enough of TCP for us to use directly in the verification of T/TCP. Murphy and Shankar in [27] also specify and verify a connection management protocol for the transport layer, but the protocol is of their own design, not TCP. In that work they also compare their protocol to TCP and they point out some problems in the TCP protocol as specified in the Internet Standard [28]. We refer to these problems when we discuss related work in section 1.4.

In our formal presentation of TCP we do make some simplifications. For example, we do not include security parameters or the congestion control aspect of TCP. We also assume a client/server model which means one side is always active and the other passive, whereas in full TCP either side can initiate communication. However, even with these simplifications, we know of no other work that formally verifies TCP in the level of generality that we present.

1.2.1 The specification of the problem

The informal specification of TCP [28, 30] is quite complicated, and an important contribution of this work is the presentation of a precise specification of the reliable transport level problem that TCP is designed to solve. The specification can be viewed as a "black box", which has a user interface that gets all the inputs that the protocol receives and sends out all the outputs that we want the protocol to produce. The specification defines a relationship on the inputs and outputs that gives precisely the desired behavior any protocol solving the problem should have.

1.2.2 Verification of TCP

Our verification of TCP has two parts. First we assume that both the client and server of TCP have unbounded counters that are stable, that is, they do not lose their values as a result of crashes. We show that this version of TCP implements our specification. Next we model TCP with bounded counters. In order for TCP with bounded counters to satisfy our specification we need to assume certain properties about the bounded counters, and the protocol has to now observe some timing restrictions. These restrictions take two basic forms. Either a host has to wait a certain period of time before performing certain actions, or a host times out and closes if it waits too long for a reply to a message.

1.2.3 Verification of T/TCP

After specifying the problem and formally verifying both versions of TCP, the next step in our verification of T/TCP was to show that it implements TCP. As described in [8, 6, 7], T/TCP does not require timeouts if responses are not received within specified time bounds. We observed that under certain circumstances T/TCP without timeouts does not behave the way TCP does, and in fact does not satisfy the specification we have for TCP. More specifically, when there is a crash T/TCP may deliver duplicate data. The fact that T/TCP can deliver duplicate data after crashes was shown by Shankar and Lee in [33]. However, the designers of T/TCP and other network protocol designers that we spoke to do not seem to think that this behavior of T/TCP is catastrophic, and they think that it may be acceptable for some situations. Therefore, in our work we formulate a weaker specification that allows these behaviors, and prove that T/TCP satisfies this specification.

In [33] Shankar and Lee present timing constraints which when incorporated into T/TCP prevents the delivery of duplicate data after a crash. Their work indicates that with these timing constraints T/TCP may satisfy our stronger specification[1]. However, while TCP needs only unbounded and stable counters, or more generally stable and infinite sets of unique identifiers (uid's) to satisfy the stronger specification, T/TCP even with stable and infinite sets of uid's still requires timing information in order to have fast transactions and still satisfy the stronger specification.

1.2.4 Impossibility result

In fact we are able to prove a more general impossibility result about protocols that try to achieve fast transactions and reliable data streaming. We first present a formal model for systems for which the impossibility result holds. This model differs from the other models used for the verifications in the thesis in that *liveness* issues that cannot be expressed in the models used for verification must be taken into consideration for the impossibility result. After presenting the model, we describe the client and server hosts in that framework. The hosts are allowed to have infinite and stable sets of uid's. We next present a formal definition of the problem of "fast transactions" and reliable data delivery which T/TCP was designed to solve. This definition differs from the specification of the reliable transport level problem mentioned in Section 1.2.1 and presented in Chapter 4 in that it requires certain messages to be delivered within a certain time bound under certain conditions. The messages that are required to be delivered within the time bound, and the conditions under which these messages are required to be delivered, depend on local state, the occurrence of crashes, the accuracy of local clocks, and message delivery times. In Chapter 11 we precisely define the assumptions about the system and precisely define the problem. We then prove that it is impossible for any protocol to solve the problem.

1.3 Formal methods

We use invariant assertion and simulation (refinement) techniques to verify TCP and T/TCP. We use the formalization of simulations developed in [24, 26] by Lynch and Vaandrager. These methods are used for proving trace inclusion relationships between concurrent

[1]We do not verify this.

systems. The methodology is developed in the context of very simple and general automaton models for both untimed [24] and timed [26] systems. For timed systems we use a formulation of the automaton model called *General Timed Automata (GTA)* presented in [21]. For the impossibility result, we also use a special case of the GTA model called clock GTA [29], which is used to model systems with local clocks. While the simple timed automaton model is useful for proving safety and some liveness properties, for the impossibility result, we need more general liveness properties than can be handled by the simple model. In particular we want the model to have the *receptiveness* property. To get this property we use the live automaton model developed in [31]. We elaborate on the basic model and methodology in Chapter 3, and on the liveness issues in Chapter 11.

1.4 Related work

Simulation techniques are known to be quite useful in the verification of concurrent systems. See, for example [1, 14, 16]. In [24, 26] Lynch and Vaandrager provide a clear framework for applying these techniques. In [18, 35], Lampson, Lynch, and Søgaard-Andersen formally verified the correctness of the five packet handshake protocol of Belsnes [4] which forms the basis of the open and close phase of TCP and ISO-TP4; and the clock synchronization protocol of Liskov, Shrira, and Wroclawski [19], using the methods developed in [24, 26, 12]. In verifying these protocols, Lampson et al. showed that simulation methods can be used to verify relatively complex and practical protocols. Our work is an extension of this work in that we use the methods to verify even more complex protocols.

Murphy and Shankar [27] also use a variation of the simulation technique to specify and verify a connection management protocol for the transport layer. They specify the connection management service for the transport layer using a state transition system and by making certain fairness assumptions. They then specify a protocol and show that it offers the service they specified for the transport layer. They use invariant assertions and a stepwise refinement heuristic [32] for the verification. In that work they compare their transport level protocol to TCP. They point out that the informal specification of TCP given in the Internet Standard [28] and the Requirements for Internet Hosts [30] does not have the right timing constraints to correctly satisfy the intended service of reliable transport level protocols. They give two of the correct timing constraints which are needed for TCP with bounded counters to behave correctly. First, they point out that timeouts are required if acknowledgments are not received within some waiting period. The informal specifications [28, 30] say these timeouts are optional. Second, they point out that the period of inactivity after crashes that is proposed in the informal specifications is not sufficient. They give a period that is sufficient. The period of inactivity after a crash that they specify requires that hosts have a bound on the maximum time it takes for a response to be generate for a received packet. The informal specification also states that such a maximum response time is optional. In our chapter on TCP with bounded counters, we show executions where violation of these timing constraints could lead to duplicate delivery of messages. We also show that some additional timeouts that Murphy and Shankar do not mention are needed for TCP to work correctly with bounded counters.

In [33] Shankar and Lee discuss what they call "minimum-latency transport protocols." These protocols are called minimum-latency because they provide the minimum latency desired for transaction-oriented applications — T/TCP fits into this category of protocols. In their paper, they define a class of caching protocols and determine the minimum counter size as a function of real-time constraints needed for such protocols. The protocols are called caching protocols because clients and servers store information in caches in these protocols. While not specifically referring to the correctness of T/TCP in their work, they do present the same scenario that we observed that may cause protocols like T/TCP to deliver the same message twice.

The impossibility result presented in the thesis is also related to the work of Shankar and Lee [33] in that they show that some timing assumptions are necessary for T/TCP and protocols like it to work correctly. The fact that these protocols require timing assumptions led us to think about whether any protocol could solve the problem of fast transactions and reliable message delivery and under what timing assumptions. Our impossibility result is in some sense a generalization of their results in that we show that if some timing assumptions do not hold, then it is impossible for any protocol to solve the problem of having fast transactions while maintaining reliable message delivery. However, our proof is for the case where the client and server have infinite and stable sets of unique identifiers, but it does not hold for the case where the client and server have unbounded counters. T/TCP and the protocols of Shankar and Lee [33] use counters. While our proof does not work for the case where protocols use counters, the counters do not seem to help in the case of T/TCP. Therefore, we believe the results hold even if protocols use counters, but we do not yet have a proof for this claim.

Most of the other theoretical work in the area of reliable message delivery has considered somewhat different problems in different settings. Afek et al. and Fekete, Lynch, Mansour, and Spinelli prove impossibility results for different types of reliable communication in a purely asynchronous setting [2, 10]. In [3], Attiya, Dolev, and Welch attain further results for the asynchronous model based on the minimum amount of information that must be maintained between connections in the presence of crashes or between active incarnations of a crashes. None of these papers examines the amount of time or number of trips across the network required to reliably deliver messages. Instead they deal with the more general question of whether reliable message delivery is possible under certain conditions regardless of number of trips across the network. However, in [17] Kleinberg, Attiya, and Lynch examine the trade-offs between message delivery and quiesce times for connection management protocols under various timing assumptions. They obtain several impossibility results for the different timing situations they consider. The impossibility results presented in that work are the closest to the impossibility result presented in this thesis. However, our result differs from their results in that we consider a more restricted problem than the problems considered in [17]. We also present a more formal development of the system for which we prove the impossibility result than the development given in [17]. We elaborate on these differences in Chapter 11 where we present the impossibility result.

1.5 Organization of the thesis

In Chapter 2 we present an informal description of TCP and T/TCP. Chapter 3 contains descriptions of the formal models and methods used in the thesis. In Chapter 4 we present two specifications for the reliable transport level problem. The first specification is the natural specification, but for reasons that we explain in the chapter, we also present an intermediate specification of the problem. In Chapter 5 we describe the formal modeling of the communication channels used for both TCP and T/TCP, and in Chapter 6 we present the formal description of TCP with unbounded counters. Chapter 7 has the proof that TCP with unbounded counters implements our specification. Chapter 8 presents TCP with bounded counters and discusses the assumptions about the counters, and the timeouts necessary for the correctness of this version of TCP. This chapter also contains the proof that this version of TCP also satisfies our specification. In Chapter 9 we present the formal description of T/TCP, and we prove that T/TCP does not implement TCP. In Chapter 10 we present a weaker specification for the transport level problem and show that T/TCP implements this weaker specification. In Chapter 11 we prove that without the correct timing assumptions it is impossible for any protocol to give the efficiency of T/TCP and still satisfy our specification. Chapter 12 contains some concluding remarks.

The thesis also contains three appendices. Appendix A contains a description of the basic notation used in the thesis and should be read before the rest of the thesis, and Appendix B and C, contain proofs required for certain results in the main part of the thesis.

Chapter 2

Informal Description of Protocols

In this chapter we present informal descriptions of TCP and T/TCP. The description of TCP we present is based on the official Internet Standard for TCP [28] and Comer's presentation in [9]. The description of T/TCP is based on Braden's and Clark's description of their design in [8]. We also present some information on the TCP/IP protocol suite to give the context in which these protocols are used. The description of the TCP/IP layering model is also based on the presentation of Comer in [9]. Our presentation in this chapter is intended to give the reader an intuitive understanding of the protocols, so that the abstract specification and the formal abstract descriptions of the protocols, which we present later, are easier to follow.

2.1 The TCP/IP layering model

In the TCP/IP layering model[1] there are four conceptual layers that build on a fifth layer of hardware. Each layer relies on the layer below it. At the highest level is the *Application Layer*. This layer consists of application programs that access services across a TCP/IP internet. The applications interact with transport level protocol(s) to send and receive data. Later in the thesis when we model the protocols we refer to "users" of the protocol. These users we refer to correspond to application programs such as telnet, email, and ftp at this level.

[1]The other main network layering model is the ISO 7-layer model.

The *Transport Layer* is the next level of the hierarchy. This layer provides communication from one application program to another. This communication is often referred to as *end-to-end*. This layer may also regulate the flow of information, and may, but does not necessarily, ensure that data arrives without error and in sequence. TCP and T/TCP belong to this layer and both protocols have the goal of ensuring reliable communication. On the other hand, the user datagram protocol (UDP) is a transport level protocol that provides unreliable delivery service. However, in this work we are only interested in transport level protocols that provide reliable data delivery service, and when we present our formal abstract specification for protocols for this layer, it is for protocols that guarantee reliable data delivery.

The next layer is the *Internet Layer* and it handles communication between machines. The internet layer accepts requests to send packets from a transport level protocol along with the destination machine, and delivers the packet to the transport level protocol at the destination machine. In between the source and destination machines the packets may be routed through intermediate links. The Internet layer is responsible for the routing of the packets. The Internet layer service is defined by the *Internet Protocol (IP)*. It is one of the two major protocols used in internetworking, the other being TCP. The IP packet delivery system is *unreliable, best-effort*, and *connectionless*. By unreliable we mean that packets can be lost, duplicated, delayed, or delivered out of order, but the service will not detect such conditions. It is called connectionless because each packet is treated independently from all others. Finally, the service is said to be best-effort because the protocol makes an earnest attempt to deliver packets, so packets get lost, delayed, or duplicated only when resources are limited or the underlying network fails. When TCP and T/TCP sends or receives packets, its interaction is with IP. In our abstract model, our unreliable channels corresponds to IP.

The fourth layer is the *Network Interface Layer* and is responsible for taking IP packets and transmitting them over a specific network. We are not concerned with this layer in our work.

2.2 Overview of TCP and T/TCP

T/TCP is an extension of TCP so both protocols are quite similar. In this section we present an overview of the general features that the protocols share. Since both protocols belong to the transport layer, they receive packets from a service that may delay the delivery of packets, deliver the packets out of order, deliver duplicates, or lose packets. On the other hand, application programs above the transport layer often need to send large volumes of data reliably from one computer to another. The sending of this data is often referred to as *data streaming* because the data can be thought of as a stream of bits. By *reliable* we mean data is delivered at-most-once and in the right order. The transfer of data is also *full duplex*; that is, data can be transferred concurrently and independently in both directions.

TCP and T/TCP are designed to provide this type of reliable data stream service. The idea behind the protocols is to give the communicating application programs the illusion that there is a circuit between them. In order to achieve this illusion a connection must be established between the two endpoints before data transfer can begin. This connection is termed a *virtual circuit connection.* The connection involves synchronizing the state at the endpoints. The endpoints of a connection are not the application programs themselves, but are instead a pair of integers of the form *(host, port)* where *host* is the IP address for a host and *port* is a port on that host. A connection is identified by its pair of endpoints. A particular connection may open and close many times. Each time the connection is opened we have what is called an *incarnation* of the connection. A single host can have several different ports that form different connections. Our work focuses on a single connection between a client (the host that initiates the connection) and a server (the host that responds). Therefore, we do not need to refer to the port numbers.

The signal from the user to the client TCP to initiate a connection is usually referred to as an *active open*, and the signal from the user to the server TCP that it can accept incoming requests to form a connection is called a *passive open*.

The unit of transfer between applications in both protocols is called a *segment*. Segments are divided into two parts — the header followed by data. The header carries control information. In practice, the IP layer may take a TCP or T/TCP segment and break it into multiple *packets*. However, in our modeling of the protocols, we assume that segments are not broken into packets, so we use the terms interchangeably to denote objects sent over the channels in an implementation. We use the terms *message* or *data* for user-meaningful data. For the purpose of our modeling we assume the header information only contains information indicating whether the segment is a SYN, FIN, or RST, segment, and a sequence number and acknowledgment number. A SYN (synchronize) indicates that the sender is sending information to try to synchronize the endpoints for the virtual circuit. A FIN (final) segment indicates that the sending host has sent its last piece of data for the current incarnation of the connection, and a RST (reset) segment indicates that the sender received a segment that is not acceptable for its current state, so the other host should reset its endpoint and try to re-synchronize. A segment may have neither of the SYN, FIN, or RST control bits. Such a segment may contain valid data and/or valid a acknowledgment. Also the SYN segment sent by the client does not contain an acknowledgment number.

Sequence numbers are used to number each SYN and FIN control signal, and each byte of data in a segment. To simplify our modeling, we assume each segment only contains one byte of data. The acknowledgment number is generated by a host when it receives a valid segment. It is the sequence number of the received segment plus one. The acknowledgment mechanism of the receiving host, along with retransmissions by the sender, ensures that segments that are lost in the network get retransmitted and eventually delivered. That is, the sender retransmits a segment after a suitable retransmission timeout (RTO) until an acknowledgment is received for that segment.

2.3 Specifics of TCP

In TCP getting synchronized states at both end-points usually requires three phases: an open phase, a bi-directional data transfer phase, and a close phase.

2.3.1 Open phase

The open phase is often referred to as the *three-way handshake protocol* because it requires the sending of three segments between the client and the server. Figure 2-1 illustrates the three-way handshake protocol. When the client receives the signal to open a connection, it chooses an initial sequence number (ISN) which will be the sequence number for the first segment it sends. It also initializes the variables it will use for the life of the connection. This set of variables is called the client's transmission control block (TCB). The server also initializes a TCB when it opens, but it does not choose an ISN until it receives an initial message from the client. Note that both the client and server must be closed in order to accept the active open and passive open signals respectively. To synchronize, the client and server must agree on their ISN's, so the client starts the three-way handshake by sending a SYN segment with its ISN. When the server receives this segment it chooses its own ISN. It also notes that the sequence number of the next segment it should receive is the ISN of the client plus one. This is the acknowledgment number that the server sends on the response segment. This response segment is also a SYN segment an includes the ISN of the server. When the client receives this return segment, it verifies that the server did receive its correct ISN, and notes the ISN of the server plus one. The final segment of the three-way handshake is the segment the client sends in response. This segment has the next sequence number for the client and an acknowledgment number of the ISN of the server plus one. When the server receives this packet, it can confirm that it has the right ISN for the client and that the client has its correct ISN. At this point both ends are synchronized and are in what is called the established state.

2.3.2 Data transfer phase

Bi-directional data transfer takes place in this state. Once the client and server agree on each other's ISN, they increment their sequence numbers for each byte of data sent. The sequence number is used to prevent the acceptance of old duplicate segments and also to order segments that might be received out of order. Initial sequence numbers are chosen such that the sequence numbers given to new segments for the new incarnation are not the same as the sequence numbers of segments from previous incarnations that might still be in the network. The acknowledgment of a segment means every segment up to that one has been successfully received. We make the simplifying assumption that every segment must get an acknowledgment before the next one is sent.

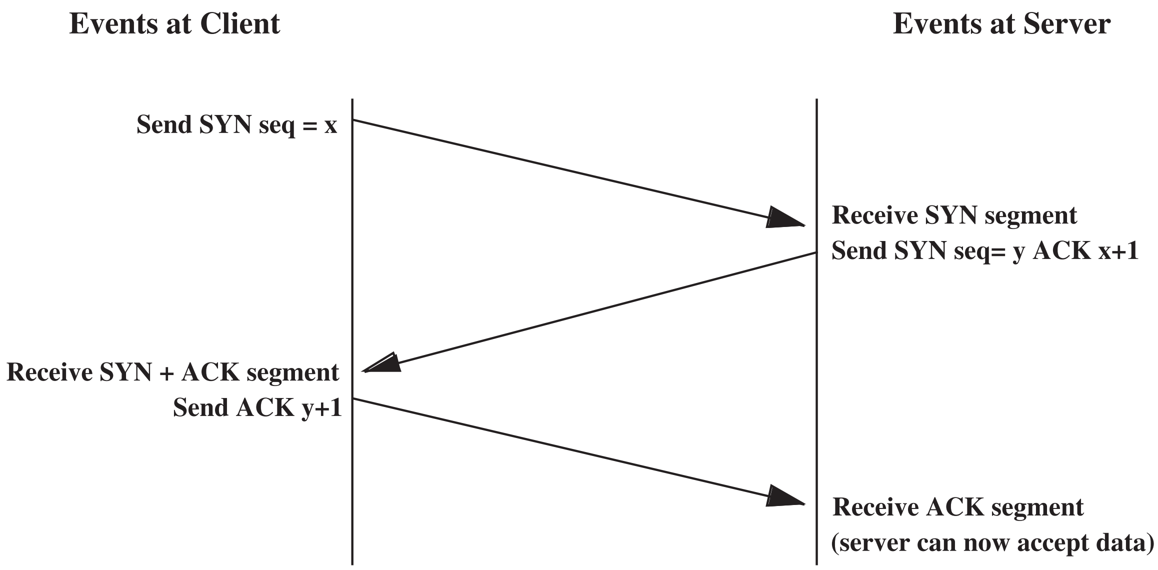

Figure 2-1: An illustration of the three-way handshake protocol. The arrows represent packets across the network.

2.3.3 Close phase

In TCP, when a host receives the signal to close from the local user it means that the user will not send any more data for that connection. However, the local user can still receive data. Therefore, both client and server must receive signals from their local users to close before a connection can close. The close phase begins when either or both hosts receive the signal to close from their local users. When a host receives the signal to close, it sends any remaining data it has to send, and then sends a FIN segment. A host that receives a FIN segment responds with an acknowledgment of the FIN segment. A host that sends a FIN segment before it receives one, when it does receive the FIN segment, waits in what is called *timed-wait* state before it closes. The duration of timed-wait state is $2 \times$ MSL (maximum segment lifetime). A host that receives a FIN segment before it sends one will close immediately upon receiving the acknowledgment for its FIN segment. It can close immediately because it must acknowledge the FIN segment it receives before it sends its FIN segment. At least one of the hosts will close from timed-wait state, and both may if they both send FIN segments before they receive one. The wait is to ensure that if a new incarnation of the same connection is started, old duplicate segments have been dropped from the network. Timed-wait state is also used to ensure the *graceful close* property. This property ensures that the host that sent the last piece of data receives confirmation that the data is received before it closes. Timed-wait state gives this property because the host(s) in that acknowledges the last piece of data waits $2 \times$ MSL before it closes. This wait is much greater than RTO, so if the acknowledgment got dropped and the host sending the last piece of data retransmitted it, the host in timed-wait state can retransmit the acknowledgment. In TCP, after a connection closes the transmission control blocks are deleted.

2.4 Specifics of T/TCP

T/TCP is designed to be a *unified transport protocol (utp)*. That is, in addition to supporting streaming as TCP does, it should also support efficient *transactions*. A transaction is typically a sequence of two messages, one in each direction, interpreted as a request and a response. The canonical example of a transaction is remote procedure call (RPC) where the call to the remote procedure is the request and the return value of the procedure is the response. TCP can be used for transactions, but because it has separate open and close phases, it is inefficient for this purpose. When large amounts of data is being sent, the bulk of the time is spent in the data transfer phase, so the overhead of the open and close phases is not significant. However, when the data being sent is just a request and a response, this overhead becomes significant. Ideally, for a transaction the time between when the client requests a service and the time it gets a response from the server should be *round trip time* (RTT) plus *server processing time* (SPT). Because of the three-way handshake in the open phase of TCP, a transaction would take a minimum of 2RTT + SPT. Another efficiency issue that comes up with transactions is that often we want to do many transactions in quick succession for the same connection. Each transaction is considered a new incarnation of the connection. Because of timed-wait state in the close phase of TCP, there has to be a wait of at least $2 \times$ MSL between transactions in TCP. The goal of the design for T/TCP is to change TCP so that the open and close phases do not make it inefficient for transactions, while maintaining the things that make it good for data streaming.

T/TCP employs two optimizations to deal with these inefficiencies. These two optimizations essentially incorporate two techniques from the implementation of RPC by Birrell and

Nelson [5] into TCP. The first optimization, known as TCP Accelerated Open (TAO), eliminates the need for the three-way handshake protocol at the opening phase of communication for most instances. Figure 2-2 illustrates the TAO mechanism. This optimization is accomplished by using a dual monotonic number scheme. That is, in addition to the sequence number, each packet carries a second number that is constant during a single incarnation and increases monotonically for each new incarnation. This second number is called a *connection count*, and it identifies each incarnation for the particular connection. Each client host has a connection count generator which is incremented every time a new incarnation for any connection from that host is initiated. Recall that a single host may have several different connections emanating from it. Associated with each connection endpoint at the client is a persistent cache value of the last connection count sent for that connection. Persistent state is state that is kept after a connection closes, but is volatile. That is, it is affected by crashes. At the server endpoint of the connection a persistent copy of the last connection count received from the client is also cached. Therefore, when a client wants to start a new incarnation of a connection, the connection count generator is incremented and the client sends the incremented connection count with the initial SYN packet containing the request data. When the server receives this packet, it checks that the connection count is greater than the last connection count it received for the connection, and can immediately accept the new data if it is. The server responds with a packet that contains response data and an echo of the client's connection count. The client uses the echoed value to determine if the response is valid. With TAO, a transaction can be carried out in one round trip across the network.

When a server crashes and recovers, it might have the cache value of the last connection count it received undefined. Thus, upon receiving the first open request from a client, it performs a three-way handshake to validate the received segment and if it is valid, updates the connection count received value to the current connection count value sent by the client.

T/TCP also uses sequence numbers to order data, but since the initial sequence number is not needed to distinguish data from different incarnations, the initial sequence number can always start at one.

The second optimization is used to shorten the close phase of TCP. Specifically, the

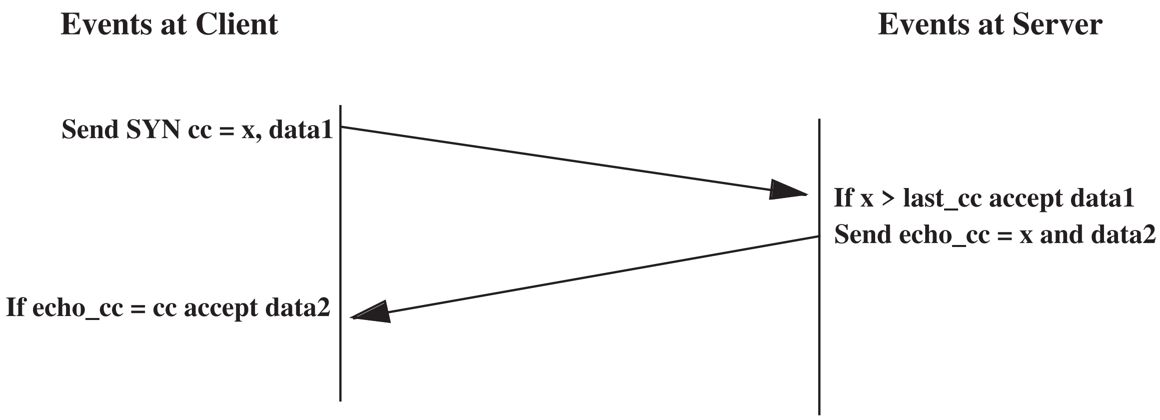

Figure 2-2: An illustration of TCP accelerate open, where data1 is the request data and data2 is the response data.

optimization reduces the mandatory wait between successive incarnations. The reduction is achieved by allowing active and passive opens to truncate timed-wait state. Thus, if a client while in timed-wait state gets an active open signal to initiate a new incarnation for the same connection, it immediately ends timed-wait state and initiates the new connection without explicitly closing the old incarnation. On the server side a passive open signal is accepted from any of the "normal" states the server could be in while the client is in timed-wait state. When the server accepts this passive open signal, it goes to a type of "bridge" state between the previous incarnation and the new one that is being established. For example, in the situation where the server receives the passive open signal while it is in a state in which it is expecting an acknowledgment of a FIN segment, if it instead receives a new SYN segment from the client, the server uses the SYN as an implicit acknowledgment of the FIN and also has an explicit signal to start a new incarnation. Thus, in T/TCP overlap of action and states for consecutive incarnations of a connection is allowed. This overlap permits rapid successive transactions for the same connection.

2.5 Other simplifications

One important aspect of both TCP and T/TCP that we do not deal with in our work is the *sliding window* mechanism. The basic idea of the sliding window mechanism is to allow several packets (the number is determined by the window size) to be sent before an acknowledgment is required. The acknowledgment of a particular packet counts as an acknowledgment of all the unacknowledged packet sent before it. When an acknowledgment is received, the sender "slides" the window forward so that it covers only unacknowledged packets and packets to be sent. In both protocols the window size is adjusted based on how fast the network can transfer packets. With the sliding window mechanism the protocols obtain much greater throughput than a protocol where every message needs an acknowledgment. To incorporate the sliding window mechanism into the basic versions of TCP or T/TCP basically requires the protocols to perform a lot more bookkeeping. In our work we are primarily concerned with the reliability of data streaming, not throughput, so we choose the simpler version of the protocols where the window size is one. If we included the sliding window mechanism in the protocols, we believe the extra bookkeeping would further complicate our proofs by requiring us to keep track of additional little details, but that conceptually the proof would not change significantly.

Chapter 3

Formal Models and Techniques

In this chapter we present definitions of the simple state machine models we use to describe the specifications and protocols in the thesis. We also present the proof techniques we use for formally verifying that the protocols implement the specification. These techniques are *invariant assertions* and *simulations*.

3.1 Automaton models

An automaton is a simple state machine or labeled transition system. We present automaton models for both untimed and timed systems, and we also state properties of these models and define operations on them.

3.1.1 Untimed automaton

The formal model we use to represent untimed systems is the *Safe I/O Automaton* model of Søgaard-Andersen et al. in [35]. This model is the same as the *I/O Automaton* model of Lynch and Tuttle [22] except it does not have the fifth component of the Lynch/Tuttle model. That fifth component is a partition of the locally controlled actions into countably many equivalence classes. It is used to define fairness conditions on an execution of the automaton. In this work we refer to the untimed model as *automaton*.

Definition 3.1 (Automaton)

An automaton, A, consists of four components:

1. A set *states(A)* of states.

2. A nonempty set *start(A)* $\subseteq$ of *states(A)*.

3. An action signature *sig(A)* = (*in(A)*, *out(A)*, and *int(A)*) a partition of the set of actions into *input actions*, *output actions* and *internal actions* respectively. The union of the *in(A)* and *out(A)* we denote as *ext(A)* the set of external actions. We denote by *local(A)* the set *out(A)* $\cup$ *int(A)* the set of locally-controlled actions, and by *acts(A)* the set *ext(A)* $\cup$ *int(A)*.

4. A transition relation *steps(A)* $\subseteq$*states(A)* $\times$ *acts(A)* $\times$ *states(A)* with the property that for every state s and input action a there is a transition (s, a, s') in *steps(A)*. A is said to be *input-enabled.* $\square$

An action a is *enabled* in a state s if there exists a state s' such that (s, a, s') is a step, that is, $(s, a, s') \in$ *steps(A).* When an automaton 'runs', it generates a string representing an execution of the system the automaton models. An *execution fragment* α of automaton A is a finite or infinite sequence, $s_0, a_1, s_1, a_2, \ldots$, of alternating states and actions of A starting in a state, and if the execution fragment is finite, ending in a state such that (s_i, a_{i+1}, s_{i+1}) is a step of A for every i. We denote by *fstate(α)* the first state of the execution fragment, and if it is finite *lstate(α)* denotes the last state. We denote by *frag***(A)*, *frag*$^\omega$*(A)*, *frag(A)* the sets of finite, infinite, and all execution fragments of A respectively. An *execution* is an execution fragment beginning with a start state. Denote by *exec***(A)*, *exec*$^\omega$*(A)*, *exec(A)* the sets of finite, infinite, and all executions of A respectively. A state s is said to be *reachable* of there exists a finite execution of A that ends in s.

A finite execution fragment $\alpha_1 = s_0, a_1, s_1, \ldots, a_n, s_n$ of A and an execution fragment $\alpha_2 = s_n, a_{n+1}, s_{n+1}, \ldots$ of A can be *concatenated.* In this case the concatenation, written as $\alpha_1 \cdot \alpha_2$, is the execution fragment $s_0, a_1, s_1, \ldots, a_n, s_n, a_{n+1}, s_{n+1}, \ldots$.

If γ is a sequence of actions, then $\hat{\gamma}$ is the sequence obtained by deleting all the internal actions of γ. We denote the empty sequence as λ. Suppose $\alpha = s_0, a_1, s_1, a_2, \ldots$ is an execution fragment of A. Let γ be the sequence consisting of the actions $a_1, a_2 \ldots$. Then $trace_A(\alpha)$ or $trace(\alpha)$ if A is clear, is defined to be the sequence $\hat{\gamma}$. That is, $trace(\alpha)$ is the subsequence of α consisting of only the external actions. We say that β is a trace of A if there exists an execution α of A with $trace(\alpha) = \beta$. We write $trace^*(A)$, $trace^\omega(A)$, $trace(A)$ for the sets of finite, infinite, and all traces of A respectively. Note that an infinite execution might have a finite trace.

In specifying a complex distributed system, it is useful to be able to specify each process individually and then obtain a specification of the entire system as the *parallel composition* of the specifications of the processes. The parallel composition operator "$\|$" in this model uses a synchronization style where automata synchronize on their common actions and evolve independently on the others. The parallel composition operator is defined only for *compatible* automata. Compatibility requires that each action be an output of at most one automaton. Furthermore, to avoid action name clashes, compatibility requires that internal action names be unique.

Definition 3.2 (Parallel composition of automata)
Automata $A_1, \ldots, A_n$ are *compatible* if for all $1 \leq i, j \leq n$ with $i \neq j$

1. $out(A_i) \cap out(A_j) = \emptyset$

2. $int(A_i) \cap acts(A_j) = \emptyset$

The *parallel composition* $A_1 \| \cdots \| A_n$ of compatible automata $A_1, \ldots, A_n$ is the automata A such that

1. $states(A) = states(A_1) \times \cdots \times states(A_n)$

2. $start(A) = start(A_1) \times \cdots \times start(A_n)$

3. $out(A) = out(A_1) \cup \cdots \cup out(A_n)$

4. $in(A) = (in(A_1) \cup \cdots \cup in(A_n)) \setminus out(A)$

5. $int(A) = int(A_1) \cup \cdots \cup int(A_n)$

6. $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in steps(A)$ iff for all $1 \leq i \leq n$

   (a) if $a \in acts(A_i)$ then $(s_i, a, s'_i) \in steps(A_i)$

   (b) if $a \notin acts(A_i)$ then $s_i =, s'_i$ $\square$

Parallel composition is typically used to build complex systems based on simpler components. Some actions are meant to represent internal communications between the subcomponents of the complex system. The *action hiding* operator "\" allows us to change some external actions into internal ones.

Definition 3.3 (Action hiding)

Let A be an automaton and let $\mathcal{A}$ be a set of actions such that $\mathcal{A} \subseteq \mathit{local}(A)$. Then define $A \setminus \mathcal{A}$ to be the automaton such that

1. $\mathit{states}(A \setminus \mathcal{A}) = \mathit{states}(A)$
2. $\mathit{start}(A \setminus \mathcal{A}) = \mathit{start}(A)$
3. $\mathit{in}(A \setminus \mathcal{A}) = \mathit{in}(A)$
4. $\mathit{out}(A \setminus \mathcal{A}) = \mathit{out}(A) \setminus \mathcal{A}$
5. $\mathit{int}(A \setminus \mathcal{A}) = \mathit{int}(A) \cup \mathcal{A}$
6. $\mathit{steps}(A \setminus \mathcal{A}) = \mathit{steps}(A)$ □

Another operation on automaton is *action renaming*. Action renaming can be used to resolve name clashes that lead to incompatibilities in Definition 3.2.

Definition 3.4 (Action renaming)

A mapping ρ from actions to actions is *applicable* to an automaton A if it is injective and $\mathit{acts}(A) \subseteq \mathit{dom}(\rho)$. Given an automaton and a mapping ρ applicable to A, we define $\rho(A)$ to be the automaton such that

1. $\mathit{states}(\rho(A)) = \mathit{states}(A)$
2. $\mathit{start}(\rho(A)) = \mathit{start}(A)$
3. $\mathit{in}(\rho(A)) = \rho(\mathit{in}(A))$
4. $\mathit{out}(\rho(A)) = \rho(\mathit{out}(A))$
5. $\mathit{int}(\rho(A)) = \rho(\mathit{int}(A))$
6. $\mathit{steps}(\rho(A)) = \{(s, \rho(a), s') \mid (s, a, s') \in \mathit{steps}(A)\}$ □

Correctness

The notion of correct implementation of automata is based on trace inclusion. That is, an automaton A is said to *implement* an automaton B with the same input and output actions if all traces of A are also traces of B. This correctness notion ensures that whatever A does, B could have done the same. That is, A does nothing wrong which means A satisfies the safety requirements specified by B.

In this work we do not verify liveness properties. Therefore, our notion of correctness does not guarantee that A does anything at all. However, since the focus of this work is to verify specific implementations that we know do something, our goal is to prove that they do nothing wrong.

Given two automata A and B such that $in(A) = in(B)$ and $out(A) = out(B)$, we say A implements B if and only if $traces(A) \subseteq traces(B)$ which we write as $A \sqsubseteq B$.

3.1.2 General Timed Automaton

In this section we present the model we use for describing systems that use time. This model, *general timed automaton (GTA)*, is the one described by Lynch in [21], and we repeat the definitions here for completeness. The notion of timed executions and timed traces are also the same as the definitions of [21]. The model is based on the timed automaton model of Lynch and Vaandrager [26]. A slight variation of the model in [26] is referred to by Segala et al. in [31] as *safe timed I/O automaton*. In [31] the safe timed I/O automaton is used as part of new I/O automaton model that can be used to express and prove general liveness properties. For most of this thesis we are not concerned with the issue of liveness. However, for the results in Chapter 11 we will need liveness properties and we will discuss the model of [31] in that chapter.

The definition of *general timed automaton* is similar to Definition 3.1, except that its set of *actions* includes special *time-passage actions* $\nu(t)$, $t \in R^+$, where R^+ is the set of positive reals. The time-passage action $\nu(t)$ denotes the passage of time by an amount t.

Definition 3.5 (General Timed Automaton (GTA))
A GTA A consists of four components:

1. A set *states(A)* of states.

2. A nonempty set *start(A)* $\subseteq$ of *states(A)*.

3. A timed action signature *t-sig(A)* = (*in(A)*, *out(A)*, *int(A)*, *time-passage(A)*) a partition of the set of actions into *input actions*, *output actions*, *internal actions*, and *time-passage actions* respectively. The union of the *in(A)* and *out(A)* is denoted as *vis(A)* the set of *visible actions*. The set of *external actions* $ext(A)$ is defined to be the set $vis(A) \cup$ *time-passage(A)*. The *discrete actions, disc(A)* is the visible and internal actions, *vis(A)* $\cup$ *int(A)*. The set of *locally-controlled actions*, *out(A)* $\cup$ *int(A)*, is denoted by *local(A)*, and the set of all actions is denoted by *acts(A)*.

4. A transition relation *steps(A)* $\subseteq$*states(A)* $\times$ *acts(A)* $\times$ *states(A)*. □

GTA's are also input enabled and a GTA A is required to satisfy the following two axioms.

A1: If $(s, \nu(t), s') \in steps(A)$ and $(s', \nu(t'), s'') \in steps(A)$, then $(s, \nu(t+t'), s'') \in steps(A)$.

A2: If $(s, \nu(t), s') \in steps(A)$ and $0 < t' < t$, then there is a state s'' such that $(s, \nu(t'), s'')$ and $(s'', \nu(t-t'), s')$ are in *steps(A)*.

Axiom **A1** allows repeated time-passage steps to be combined into one step, and Axiom **A2** is a kind of converse to **A1**; it says that any time-passage step can be split in two.

Timed executions

A *timed execution fragment* of a GTA A is a finite or infinite alternating sequence $\alpha = s_0, a_1, s_1, a_2, \ldots$, where, the s's are states of A and the a's are actions (either input, output, internal, or time-passage) of A, and (s_i, a_{i+1}, s_{i+1}) is a step of A for every i. The sequence must begin with a state, and if it is finite must end with a state. As with the untimed automaton model, we denote by *fstate*(α) the first state of the timed execution fragment α, and if it is finite *lstate*(α) denotes the last state. We denote by *t-frag***(A)*, *t-frag*$^\omega$*(A)*, *t-frag(A)* the sets of finite, infinite, and all timed execution fragments of A respectively. A *timed execution* is a timed execution fragment beginning with a start state. Denote by *t-exec***(A)*, *t-exec*$^\omega$*(A)*, *t-exec(A)* the sets of finite, infinite, and all executions of A respectively. The concatenation of timed execution fragments is the same as for untimed execution fragments.

If α is a timed execution fragment and a_i is any discrete action in α, then we say the *time of occurrence* of a_i is the sum of all the reals in the time-passage actions preceding a_i in α. For a timed execution fragment α define *ltime*(α), the *last time* of α, to be the supremum of the sum of all the time passage actions in α.

Timed executions and timed execution fragments can be partitioned into *finite*, *admissible*, and *Zeno* timed executions and timed execution fragments. A timed execution (fragment) α is defined to be, *finite* if it is a finite sequence and *ltime*(α) is finite. It is defined to be *admissible* if $ltime(\alpha) = \infty$, and it is defined to be *Zeno* if it is neither finite or admissible. Denote by $\textit{t-frag}^Z(A)$ and $\textit{t-exec}^Z(A)$ the sets of Zeno timed execution fragments and timed executions of a GTA A respectively.

The *timed trace* of a timed execution fragment α, written *t-trace*(α), is the pair consisting of the sequence of visible actions of α paired with their time of occurrence and the last time of α. More formally, if $\alpha = s_0, a_1, s_1, a_2, \ldots$ is a timed execution fragment of a GTA A. For each $a_i \notin$ *time-passage(A)*, let t_i be its time of occurrence. Now let $\delta = (a_1, t_1)(a_2, t_2)\cdots$ be the sequence consisting of the non-time-passage actions in α paired with their time of occurrence. Then *t-trace*(α), is defined to be the pair[1]

$$\textit{t-trace}(\alpha) \triangleq (\delta \restriction (vis(A) \times R^{\geq 0}), ltime(\alpha))$$

If we have timed execution fragments that differ only by splitting and combing time-passage steps, then since we have Axioms **A1** and **A2** there is really not much difference between such timed executions fragments. Therefore, an equivalence relation can be defined on timed execution fragments that says they are the same except for time-passage. More formally, we say that one timed execution fragment α is a *time-passage refinement* of another timed execution fragment α' provided that α and α' are identical except for the fact that in α, some of the time-passage steps of α' are replaced with finite sequences of time-passage

[1] Recall that the symbol $\restriction$ denotes the projection of a sequence on a subset of the domain of its elements.

steps, with the same initial and final states and the same total amount of time passage. We say timed execution fragments α and α' are *time-passage equivalent* if they have a common time-passage refinement.

The definitions for *parallel composition*, *action hiding*, and *action renaming* also apply to GTA.

Correctness

The correctness notion for timed automata is similar to our notion for untimed automata, the difference being that it is based on timed traces. Thus, given two timed automata A and B such that *in(A)* = *in(B)* and *out(A)* = *out(B)*, we say A implements B if and only if *t-traces(A)* $\subseteq$ *t-traces(B)* which we write as $A \sqsubseteq_t B$.

3.1.3 Embedding results

We give specifications for a reliable transport level service using untimed automata because the problem description does not require the use of time. However, TCP and T/TCP use time, so we present them as timed automata. The methods we use do not allow us to show trace inclusion between timed and untimed systems, so we cannot directly show TCP or T/TCP implements our untimed specifications. The same issue comes up in the work of Søgaard-Andersen et al. [35] and they use the *patient* operator that converts an untimed automaton to a timed automaton by adding arbitrary time passage steps. They then show that the timed traces of the implementation are a subset of the timed traces of the *patient* specification. The patient operator and the embedding theorem which we present below are developed in the work of Segala et al. [31] to handle these types of untimed specification/timed implementation situations. We present their definition of the patient operator below.

Definition 3.6 (Patient automaton)
Let A be an automaton such that $\{\nu(t) \mid t \in R^+\} \cap acts(A) = \emptyset$. Then define *patient(A)* to be the GTA with

1. *states(patient(A))* = *states(A)* $\times R^{\geq 0}$

2. $start(patient(A)) = start(A) \times \{0\}$

3. $ext(patient(A)) = ext(A) \cup \{\nu(t) \mid t \in R^+\}$

4. $in(patient(A)) = in(A)$

5. $out(patient(A)) = out(A)$

6. $int(patient(A)) = int(A)$

7. *steps(patient(A))* consists of the steps

   (a) $\{((s,t), a, (s',t)) \mid (s,a,s') \in steps(A)\}$

   (b) $\{((s,t), \nu(t'), (s',t'')) \mid t + t' = t''\}$ □

For technical reasons which we discuss in Chapter 4, we have two versions of the specification (both untimed) which we call S and D. We show D implements S, and in Chapter 6 we show that the low level protocol (TCP) implements *patient(D)*. Using the fact that TCP implements *patient(D)*, we would like to say that TCP also implements *patient(S)*. In order to say this we need the *Embedding Theorem* of Segala et al. [31] which states that untimed protocol A implements untimed protocol B if and only if *patient(A)* implements *patient(B)*. Formally this is stated as:

Theorem 3.1
Let A and B be automata such that $\{\nu(t) \mid t \in R^+\} \cap (acts(A) \cup acts(B)) = \emptyset$. Then $A \sqsubseteq B$ iff $patient(A) \sqsubseteq_t patient(B)$. ■

This concludes the introduction to the basic models for untimed and timed systems we use in this work.

3.2 Verification Techniques

The techniques we use in this work are *invariant assertions* and *simulation methods*. These methods are used for proving trace inclusion relationships between concurrent systems. We describe these methods for both the untimed and timed automata models. For the untimed setting, the presentation we give here is based on the description of the techniques given

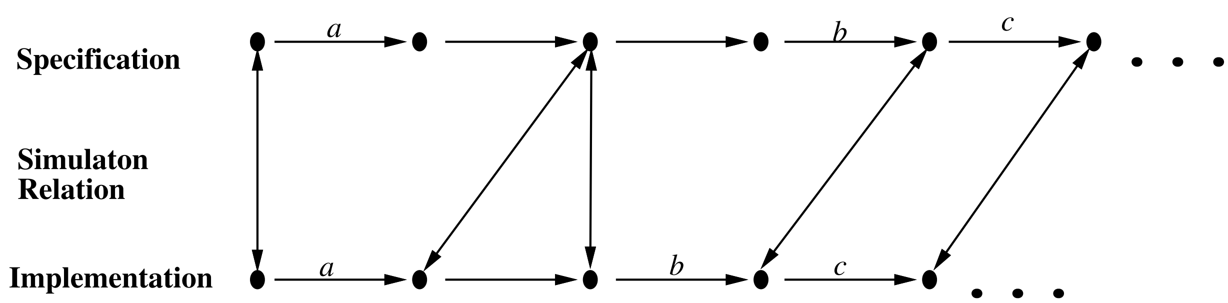

Figure 3-1: *Example of a simulation. The horizontal arrows represent steps of the automaton. The labels a, b and c are external actions and the unlabeled arrows represent steps generated by internal actions.*

by Søgaard-Andersen et al. in [35]. That description is in turn based on the work of Lynch and Vaandrager in [24]. For the general timed automata model, we use the formalization of simulations developed in [26] by Lynch and Vaandrager.

3.2.1 Untimed Automata

In this section we present a number simulation techniques for untimed automata. We also present a description of *auxiliary variables* which are used to augment the simulation techniques.

Simulation techniques

Let A be an untimed automaton representing a concrete implementation of a protocol and B an untimed automaton representing an abstract specification of the protocol. If A and B have the same input and output actions, a simulation from A to B is a relation between states of A and states of B such that certain conditions hold. The conditions that hold depend on the which of the two simulation methods we use. The methods are *forward* simulations (a special case of which is a *refinement mapping*) and *backward* simulations. Two basic conditions that must be satisfied are, first, the start states of the two automata must be related in a certain way, and second, each step of the implementation must "simulate" some sequence of steps in the specification. That is, for each step in the implementation, there must exist a sequence of steps in the specification between states related by the simulation relation to the pre and post-state of the implementation step, such that the sequence of specification steps contains exactly the same external actions as the implementation step. How the sequence of specification steps is chosen depends on the simulation method we use. Figure 3-1 is illustrates the second condition. In the figure, the external action a of the implementation is simulated by the same external action and an internal action in the specification; the next action of the implementation is internal and is simulated by the empty action in the specification; b is simulated by an internal action and b; and finally c is simulated by c.

Below, refinement mappings, forward simulations, and backward simulations are formally defined. Further results about these simulations are presented in [24]. Since we only need to consider the reachable states of specifications and implementations, we assert invariants on the states of the automata to restrict the states that need to be considered. An invariant on an automaton A is defined to be a state formula over A that is satisfied by (at least) all the reachable states of A. Another way to say this is that an invariant on A is a property that is true for all reachable states of A.

In the definitions below and throughout the thesis we use the following notational convention: if R is a relation over $S_1 \times S_2$ and $s_1 \in S_1$, then $R[s_1]$ denotes the set $\{s_2 \in S_2 \mid (s_1, s_2) \in R\}$.

Definition 3.7 (Refinement mapping)

Let A and B be automata with $in(A) = in(B)$ and $out(A) = out(B)$ and with the invariants I_A and I_B respectively. A *refinement mapping* from A to B, with respect to I_A and I_B, is a function r from $states(A)$ to $states(B)$ that satisfies:

1. If $s \in start(A)$ then $r(s) \in start(B)$

2. If $(s, a, s') \in steps(A)$, $s, s' \in I_A$, and $r(s) \in I_B$, then there exists $\alpha \in frag^*(B)$ with $fstate(\alpha) = r(s)$, $lstate(\alpha) = r(s')$, and $trace(\alpha) = trace(s, a, s')$.

We write $A \leq_R B$ if there exists a refinement mapping from A to B with respect to some invariants I_A and I_B, and if r is such a mapping we write $A \leq_R B$ via r. □

Theorem 3.2 (Soundness of refinement mapping)

$A \leq_R B \Rightarrow A \sqsubseteq B$.

Proof: The proof of this theorem is presented in [24]. ■

A forward simulation is a generalization of a refinement mapping and is defined below. Instead of having a mapping, a relation is defined on the states of the specification and implementation.

Definition 3.8 (Forward simulation)

Let A and B be automata with *in(A) = in(B)* and *out(A) = out(B)* and with the invariants I_A and I_B respectively. A *forward simulation* from A to B, with respect to I_A and I_B, is a relation f over *states(A)* $\times$ *states(B)* that satisfies:

1. If $s \in$ *start(A)* then $f[s] \cap$ *start(B)* $\neq \emptyset$.

2. If $(s, a, s') \in$ *steps(A)*, $s, s' \in I_A$, and $u \in f[s] \cap I_B$, then there exists $\alpha \in$ *frag*(B)* with *fstate*$(\alpha) = u$, *lstate*$(\alpha) \in f[s']$, and *trace*$(\alpha) =$ *trace*(s, a, s').

We write $A \leq_F B$ if there exists a forward simulation form A to B with respect to some invariants I_A and I_B, and $A \leq_F B$ via f if f is such a simulation. □

Theorem 3.3 (Soundness of forward simulation)

$A \leq_F B \Rightarrow A \sqsubseteq B$.

Proof: The proof of this theorem is presented in [15, 22, 36]. ■

The word "forward" in a forward simulation refers to the fact that a high-level sequence of steps is constructed from any possible pre-state in a forward direction toward the set of possible post-states.

On the other hand, in a backward simulation the steps are constructed in a backward direction. That is, a sequence of high-level steps ending in any state related to the low-level post-state and starting in some some state related to the low-level pre-state has to be found. Before we define a backward simulation we make the auxiliary definition of image-finiteness.

Definition 3.9 (Image-finiteness)

A relation R over $S_1 \times S_2$ is *image-finite* if for each $s_1 \in S_1$, $R[s_1]$ is a finite set. □

Definition 3.10 (Backward simulation)

Let A and B be automata with *in(A) = in(B)* and *out(A) = out(B)* and with the invariants I_A and I_B respectively. A *backward simulation* from A to B, with respect to I_A and I_B, is a relation b over *states(A)* $\times$ *states(B)* that satisfies:

1. If $s \in I_A$ then $b[s] \cap I_B \neq \emptyset$.

2. If $s \in$ *start(A)* then $b[s] \cap I_B \subseteq$ *start(B)*.

3. If $(s, a, s') \in$ *steps(A)*, $s, s' \in I_A$, and $u' \in b[s'] \cap I_B$, then there exists $\alpha \in$ *frag***(B)* with *lstate*$(\alpha) = u'$, *fstate*$(\alpha) \in b[s] \cap I_B$, and *trace*$(\alpha) =$ *trace*(s, a, s').

We write $A \leq_B B$ if there exists a backward simulation form A to B with respect to some invariants I_A and I_B. Furthermore, if the simulation is image-finite, we write $A \leq_{iB} B$. If b is a backward simulation from A to B with respect to some invariants I_A and I_B, we write $A \leq_B B$ (or $A \leq_{iB} B$ when b is image-finite) via b. □

Theorem 3.4 (Soundness of backward simulation)
$A \leq_{iB} B \Rightarrow A \sqsubseteq B$.

Proof: The proof of this theorem is presented in [24]. ■

Auxiliary variables

In [1] Abadi and Lamport show that in some instances even though it is not possible to find a mapping from A to B, by adding appropriate *auxiliary variables* to A to get A_{aux}, a refinement mapping can be found from A_{aux} to B. Since A can be shown to be equivalent to A_{aux} (that is, to have the same set of traces), the soundness of refinement mapping implies A implements B. There are two types of auxiliary variables, *history variables* and *prophecy variables*. We only consider history variables in this work. History variables are allowed to record the past history of the system. Thus, history variables are allowed in each step to be assigned a value based on all variables in the system, but must not affect the enabledness of actions or the changes made to the other (ordinary) variables. Rules for syntactically adding history variables to a system are easy to define and are presented by Søgaard-Andersen et al. in [35]. The reader is referred to [35] for more details on auxiliary variables. In [24] and [26] *history* and *prophecy relations* are defined, and shown to be abstract versions of history and prophecy variables.

We use the following theorem in the proofs later in this work.

Theorem 3.5

Let A^h be obtained from untimed automaton A by adding history variables, and let B be an untimed automaton. Then, if $A^h \leq_R B$ then $A \sqsubseteq B$.

Proof: The proof of this theorem is presented in [1, 24]. The proof in [24] is presented in terms of history relations. ■

3.2.2 Timed simulations

Simulation between timed automata is similar to simulation between untimed automata, except now we are concerned with timed traces.

Definition 3.11 (Timed refinement mapping)

Let A and B be general timed automata with *in(A)* = *in(B)* and *out(A)* = *out(B)* and with the invariants I_A and I_B respectively. A *timed refinement mapping* from A to B, with respect to I_A and I_B, is a function r from *states(A)* to *states(B)* that satisfies:

1. If $s \in$ *start(A)* then $r(s) \in$ *start(B)*
2. If $(s, a, s') \in$ *steps(A)*, $s, s' \in I_A$, and $r(s) \in I_B$, then there exists $\alpha \in$ *t-frag*(B)* with *fstate*$(\alpha) = r(s)$, *lstate*$(\alpha) = r(s')$, and *t-trace*(α) = *t-trace*(s, a, s').

We write $A \leq_R^t B$ if there exists a timed refinement mapping from A to B with respect to some invariants I_A and I_B, and if r is such a mapping we write $A \leq_R^t B$ via r. □

Theorem 3.6 (Soundness of timed refinement mapping)

$A \leq_R^t B \Rightarrow A \sqsubseteq_t B$.

Proof: The proof of this theorem is presented in [26]. ■

Definition 3.12 (Timed forward simulation)

Let A and B be general timed automata with *in(A)* = *in(B)* and *out(A)* = *out(B)* and with the invariants I_A and I_B respectively. A *timed forward simulation* from A to B, with respect to I_A and I_B, is a relation f over *states(A)* $\times$ *states(B)* that satisfies:

1. If $s \in$ *start(A)* then $f[s] \cap$ *start(B)* $\neq \emptyset$.
2. If $(s, a, s') \in$ *steps(A)*, $s, s' \in I_A$, and $u \in f[s] \cap I_B$, then there exists $\alpha \in$ *t-frag*(B)* with *fstate*$(\alpha) = u$, *lstate*$(\alpha) \in f[s']$, and *t-trace*(α) = *t-trace*(s, a, s').

We write $A \leq_F^t B$ if there exists a timed forward simulation form A to B with respect to some invariants I_A and I_B, and $A \leq_F^t B$ via f if f is such a simulation. $\square$

Theorem 3.7 (Soundness of forward simulation)

$A \leq_F^t B \Rightarrow A \sqsubseteq_t B$.

Proof: The proof of this theorem is presented in [26]. $\blacksquare$

Definition 3.13 (Timed backward simulation)

Let A and B be general timed automata with *in(A)* = *in(B)* and *out(A)* = *out(B)* and with the invariants I_A and I_B respectively. A *timed backward simulation* from A to B, with respect to I_A and I_B, is a relation b over *states(A)* $\times$ *states(B)* that satisfies:

1. If $s \in I_A$ then $b[s] \cap I_B \neq \emptyset$.
2. If $s \in$ *start(A)* then $b[s] \cap I_B \subseteq$ *start(B)*.
3. If $(s, a, s') \in$ *steps(A)*, $s, s' \in I_A$, and $u' \in b[s'] \cap I_B$, then there exists $\alpha \in$ *t-frag***(B)* with *lstate*$(\alpha) = u'$, *fstate*$(\alpha) \in b[s] \cap I_B$, and *t-trace*$(\alpha)$ = *t-trace*(s, a, s').

We write $A \leq_B^t B$ if there exists a timed backward simulation form A to B with respect to some invariants I_A and I_B. Furthermore, if the timed simulation is image-finite, we write $A \leq_{iB}^t B$. If b is a backward simulation from A to B with respect to some invariants I_A and I_B, we write $A \leq_B^t B$ (or $A \leq_{iB}^t B$ when b is image-finite) via b. $\square$

Theorem 3.8 (Soundness of timed backward simulation)

$A \leq_{iB}^t B \Rightarrow A \sqsubseteq_t B$.

Proof: The proof of this theorem is presented in [26]. $\blacksquare$

As is the case for untimed simulations, *history variables* can be added to general timed automata, so that a timed refinement mapping can be found. The rules for adding history variables to general timed automata are the same as in the untimed case.

Theorem 3.9

Let A^h be obtained from general timed automaton A by adding history variables, and let B be a general timed automaton. Then, if $A^h \leq_{tR} B$ then $A \sqsubseteq_t B$.

Proof: The proof of this the theorem follows from a proof about history relations for timed automata in [26]. $\blacksquare$

Chapter 4

The Abstract Specifications

In this chapter we present two abstract formal specifications of the user visible behavior for reliable transport level protocols of the TCP/IP Internet layering model. The first specification S is the more natural one, while the second specification D is needed for technical reasons. We discuss the reasons for specification D in Section 4.2. As discussed in Chapter 2, the transport level layer is responsible for reliable communication between application programs. By reliable we mean data is not duplicated, reordered, or lost (except in the case of crashes or aborts).

A specification of a problem should describe precisely the essential behavior we want the protocol solving that problem to exhibit. That is, the specification is rigorous enough to prove theorems, but is not cluttered with unnecessary details. The specification can be viewed as a "black box" which has a user interface that gets all the inputs that the protocol receives and sends out all the outputs that we want the protocol to produce. The specification defines a relationship on the inputs and outputs that gives precisely the desired behavior any protocol solving the problem should have. The user interface for TCP and our specifications, S and D, is shown in Figure 4-1.

The user interface for TCP in the Internet standard [28], has an explicit *active-open* input and separate *send-msg* and *close* inputs. We combined these actions in our specification into the single $\mathit{send\text{-}msg}_c(\mathit{open}, m, \mathit{close})$ [1] action on the client side because we want to

[1] *open* and *close* are boolean, and $m \in \mathit{Msg} \cup$ `null`, where the set *Msg* is the set of all possible finite strings over some basic message alphabet that does not include the special symbol `null`.

allow for the situation where the client side user opens the connection, sends just one message, and closes immediately. The interface where the actions are combined facilitates such a transaction without losing any of the functionality of the usual TCP interface. Braden in [7] suggests a similar interface for T/TCP. We do not combine the three actions into one action on the server side because that side is passive and cannot send any data until it has formed a connection with the client. However, we combine the *send-msg* and *close* actions to facilitate a reply message and an immediate close.

4.1 The specification S

4.1.1 Informal description of the specification S

Before we give the precise formal specification, we present an informal description of the specification, give the intuition behind our choices, and informally explain why they work. The specification we will present is loosely based on the specification given for the at-most-once message delivery problem in [35].

The first important point about the specification is that it is not distributed in the true sense even though it is presented as having a client and a server side. It is not distributed because client side variables can be read by the server side and vice versa. In addition, there are variables that can be written by either side. To capture the essence of at-most-once delivery of messages, we use FIFO queues. Data is added to the back of a queue, and removed from the front. Since the queues do not lose or duplicate data, we get the property we want. If there is a crash, then some data can be nondeterministically removed from the back of the queues.

The other significant feature we have to capture in our specification is that connections may have multiple incarnations and that data sent in each incarnation must be separated. This also means we have to capture a notion of the host being open or closed. To capture the idea of the sides opening and forming a connection, we assign id's from infinite sets to the client and the server ends when they open, and then pair them to form an association. To make sure associations are distinct, an id is never paired with more than one other id, and each id is used only once. To guarantee that each id can only be chosen once and associate with only one other id, we have two infinite sets of id's, one for the client side (CID) and the other for the server side (SID). We also have a set, which we call *assoc*, that keeps track of the associations that have already been formed.

What we have described so far makes each incarnation unique, but it still does not guarantee that data from different incarnations will be separated. To ensure the separation of this data, we have two infinite arrays of FIFO queues. The queues are indexed by the id's of the client and server. That is, the client sends data on the queue indexed by its id, and the server sends data on the queue indexed by its id. The array of queues that take data from the client to the server we call $queue_{cs}$, and the array of queues that take data from the server to the client we call $queue_{sc}$. A host can only receive data from a queue if its current id is associated with the id of the sender of the data, and since each id can only be associated with one other id, a host can only receive data from a unique incarnation during the life of that incarnation. Thus, there is no danger of receiving data from a previous incarnation.

Associated with each $queue_{cs}$ and $queue_{sc}$ are the flags $q\text{-}stat_{cs}$ and $q\text{-}stat_{sc}$ respectively. A queue's status is `dead` if it has never been used to send messages, or if it has been used and its receiving host has closed or crashed. Only queues with status `live` can have messages added and/or removed. Since we use id's to indicate an open host, we use the special value `nil` to indicate when a host is closed. That is, when a host has id value `nil`, it is closed. A host should only close, barring a crash or an abort, when it has sent all its data (it received a *close* signal from the local user) and when it has received all the data from the other host. Here we use the fact that the specification is not distributed to have the remote host determine when the other host has sent all its data. In the formal specification the internal action to close the client side is $set\text{-}nil_c(j)$, where j is the value of a server side id paired with the current client side id. Queues becomes `dead` when the receiving host crashes as a matter of definition. Conceivably, the sender could still add data to a queue when the receiver crashes, but this data can never be received, so in the specification we do not allow it to be added.

A host may also close if it opened, but did not receive any data or form an association and got the close signal from the user. In the formal specification the action to close in this

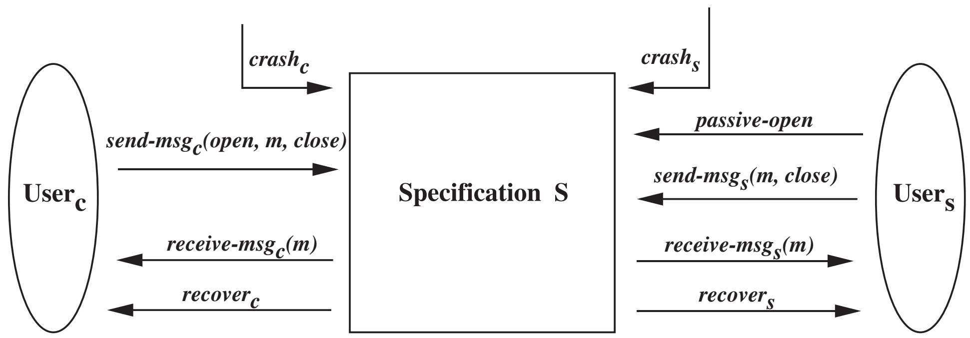

Figure 4-1: *The user interface for TCP/IP transport level protocols.*

situation is *reset-nil*. A crash is represented as an input from an external source, and is meant to indicate that something beyond the control of the specification has gone wrong. An abort, on the other hand, is an internal action that has basically the same effects as a crash. It is used to represent the fact that in low level implementations, a host my a decide to abruptly close its end of the connection without receiving any external signal to do so. This type of abrupt close usually happens in low level protocols when a host determines that something may be wrong at the other host.

4.1.2 The formal specification S

We use the untimed automaton model described in Chapter 3 to formally present the specification. We start by presenting the action signature, then the set of states with the start states, and finally we present the set of steps. The steps are presented in a *precondition, effect* style commonly used with I/O automata [22]. That is, the state during which an act is enabled is given as a precondition, and the resulting state is given by the effects of the action.

States and start states

In the specification we use the set *Msg* to represent the set of possible messages. That is, the set *Msg* is the set of all possible strings over some basic message alphabet that does not include the special symbol `null`. The symbol `null` indicates the absence of a message. We also use two sets of unique identifiers (uid's). Elements of these sets are used to uniquely identify each incarnation. The client side uses a set called *CID* and the server side uses a set called *SID*. These sets can be arbitrary, but they must be infinite and they cannot contain the special value `nil`. Elements of *CID* and *SID* are used to index the infinite arrays of queues $queue_{cs}$ and $queue_{sc}$ respectively. They are also used to index the array of flags $q\text{-}stat_{cs}$ and $q\text{-}stat_{sc}$ respectively. The first table below summarizes the type definitions, and the other two tables describe all the variables we use, and also has the initial value of each.

Type	**Description**
Msg	The set of all possible strings over some basic message alphabet that does not include the special symbol `null`.
CID	An infinite set that does not include the special value `nil`.
SID	An infinite set that does not include the special value `nil`.

Variable	**Type**	**Initially**	**Description**
id_c	$CID \cup \{\text{nil}\}$	`nil`	A unique identifier for the client side of an association pair.
id_s	$SID \cup \{\text{nil}\}$	`nil`	A unique identifier for the server side of an association pair.
choose-sid	Bool	`false`	A flag that is set to true when the server receives the signal to open. It enables the server to choose an id, and it is set to false after the id is chosen.
$mode_c$	{`active`, `inactive`}	`inactive`	The value `active` indicates that the client has received the signal to *open*, and `inactive` indicates that the client received the *close* signal or is in the initial state.
$mode_s$	{`active`, `inactive`}	`inactive`	Symmetric to $mode_c$.
$used\text{-}id_c$	2^{CID}	$\emptyset$	Set of id's already used by the client side.
$used\text{-}id_s$	2^{SID}	$\emptyset$	Set of id's already used by the server side.
$queue_{cs}$	an array indexed by *CID*, of Msg^*	$\forall\, i \in CID$, $queue_{cs}(i) = \epsilon$	An infinite array of FIFO queues. Queues from this array hold messages sent from the client side.
$queue_{sc}$	an array indexed by *SID*, of Msg^*	$\forall\, j \in SID$, $queue_{sc}(j) = \epsilon$	An infinite array of FIFO queues. Queues from this array hold messages sent from the server side.
$q\text{-}stat_{cs}$	an array indexed by *CID*, of {`dead`, `live`}	$\forall\, i \in CID$, $queue_{cs}(i) =$ `dead`	An infinite array of flags. Each flag indicates the status of the element of $queue_{cs}$ with the same index. Data can only be placed on or received from the queue if the flag is `live`.
$q\text{-}stat_{sc}$	an array indexed by *SID*, of {`dead`, `live`}	$\forall\, j \in SID$, $queue_{sc}(j) =$ `dead`	An infinite array of flags. Each flag indicates the status of the element of $queue_{sc}$ with the same index. Data can only be placed on or received from the queue if the flag is `live`.

Variable	Type	Initially	Description
$assoc$	$2^{(CID \times SID)}$	$\emptyset$	A set of pairs of id_c's and id_s's that contains each association formed between client and server.
rec_c	Bool	`false`	True if and only if the client side has crashed and not yet recovered.
rec_s	Bool	`false`	Symmetric to rec_c.
$abrt_c$	Bool	`false`	True if and only if the client side has aborted and not yet shut down.
$abrt_s$	Bool	`false`	Symmetric to $abrt_c$.

We also define two derived variables. These variables, $live\text{-}q_{cs}$ and $live\text{-}q_{sc}$, are sets that contain the indices of members of $q\text{-}stat_{cs}$ and $q\text{-}stat_{sc}$ respectively that have the value `live` in a given state.

We use a dot (.) to refer to the value of variable in a particular state. For example, $u.mode_s = \texttt{active}$ means that variable $mode_s$ has the value `active` in state u. If we do not explicitly note the state, we mean the value of the variable in any state.

$$u.live\text{-}q_{cs} \triangleq \{i \mid u.q\text{-}stat_{cs}(i) = \texttt{live} \ \wedge i \in CID\}$$

$$u.live\text{-}q_{sc} \triangleq \{i \mid u.q\text{-}stat_{sc}(j) = \texttt{live} \ \wedge j \in SID\}.$$

Action signature

Input:
$send\text{-}msg_c(open,\ m,\ close)$,
 $m \in$ Msg $\cup\{$`null`$\}$, $open,\ close \in$ Bool
$crash_c$
$passive\text{-}open$
$send\text{-}msg_s(m,\ close)$,
 $m \in$ Msg $\cup\{$`null`$\}$, $close \in$ Bool
$crash_s$

Output:
$receive\text{-}msg_c(m)$ $m \in$ Msg
$receive\text{-}msg_s(m)$ $m \in$ Msg
$recover_c$
$recover_s$

Internal:
$choose\text{-}server\text{-}id(j)$, $j \in SID$
$make\text{-}assoc(i,j)$, $i \in CID$, $j \in SID$
$set\text{-}nil_c(j)$
$set\text{-}nil_s(i)$
$reset\text{-}nil_c$
$reset\text{-}nil_s$
$abort_c$
$abort_s$
$shut\text{-}down_c$
$shut\text{-}down_s$
$lose_c(I)$
$lose_s(I)$

Steps

The steps of the automaton for the specification, S, are shown in Figures 4-2 and 4-3. In the specification, if a side has crashed or aborted, then the actions for that side, except the ones having to do with crashes or aborts should be disabled. Thus, in the steps of S, most of the client side actions have as a precondition or as a condition on the effect clause that $\neg(rec_c \vee abrt_c)$. On the server the symmetric condition is $\neg(rec_s \vee abrt_s)$.

When the *send-msg*$_c$*(open, m, close)* action is received, there are several possibilities. If *open* is true and the client is not sending or receiving messages, that is, id_c is `nil`, then the client side opens by choosing a new id_c for its half of the association pair. We use the notation $:\in$ for assigning a variable a arbitrarily chosen element of a set. Once that id is chosen, it is added to the set *used-id*$_c$ so that it will not be chosen again. Next a queue is activated to send data from the client to the server for this incarnation by setting $q\text{-}stat_{cs}(id_c)$ to `live`. To indicate that the user can send data, $mode_c$ is set to `active`. If the m in *send-msg*$_c$*(open, m, close)* is not `null` and the user can still send data ($mode_c =$ `active`), then the message is added to the back of $queue_{cs}(id_c)$ if it is `live`. This queue is the unique queue for messages associated with id_c. If *close* is true, then $mode_c$ is set to `inactive`.

On the server side, the *passive-open*, *choose-server-id(j)*, and *send-msg*$_s$*(m, close)* actions combine to do what is essentially the server side equivalent of what the *send-msg*$_c$*(open, m, close)* action does on the client side.

The *passive-open* input indicates that the server side is now able to form a connection with an open client side. However, because in some low level protocols the server does not actually choose an id until it receives a message from the client, in the specification the server does not choose an id in the *passive-open* action. Instead, it is enabled to choose one after the action is completed. When the server side receives this input, if it is not currently sending or receiving messages, that is, id_s is `nil`, it enables the choosing of an id by setting the *choose-sid* flag to `true`, and it sets $mode_s$ to `active` to indicate that the server side user may send data.

In the internal action *choose-server-id(j)*, the server nondeterministically chooses an id j from *SID* that has not already been used by the server side. This id is assigned to id_s, and is added to the set of used server id's. The queue indexed by j is also made `live`.

In the *send-msg*$_s$*(m, close)* action, if $m \neq$ `null`, it is added to the back of the queue indexed by the current id of the server. If *close* is true, $mode_s$ is set to `inactive` to indicate

send-msg$_c$(open, m, close)
Eff: if $\neg(rec_c \vee abrt_c)$ then
  if $open \wedge id_c = \texttt{nil}$ then
    $id_c :\in CID \setminus used\text{-}id_c$
    $used\text{-}id_c := used\text{-}id_c \cup \{id_c\}$
    $mode_c := \texttt{active}$
    $q\text{-}stat_{cs}(id_c) := \texttt{live}$
  if $mode_c = \texttt{active} \wedge m \neq \texttt{null} \wedge$
  $q\text{-}stat_{cs}(id_c) = \texttt{live}$ then
    $queue_{cs}(id_c) := queue_{cs}(id_c) \cdot m$
  if $close$ then $mode_c := \texttt{inactive}$

make-assoc(i,j)
Pre: $i \in used\text{-}id_c \wedge j \in used\text{-}id_s \wedge$
  $\forall\, k\ (i,k) \notin assoc \wedge \forall\, l\ (l,j) \notin assoc$
Eff: $assoc := assoc \cup \{(i,j)\}$

receive-msg$_c$(m)
Pre: $\neg(rec_c \vee abrt_c) \wedge$
  $q\text{-}stat_{sc}(j) = \texttt{live} \wedge (id_c, j) \in assoc \wedge$
  $queue_{sc}(j) \neq \epsilon \wedge head(queue_{sc}(j)) = m$
Eff: $queue_{sc}(j) := tail(queue_{sc}(j))$

set-nil$_c$(j)
Pre: $\neg(rec_c \vee abrt_c) \wedge$
  $id_c \neq \texttt{nil} \wedge mode_c = \texttt{inactive} \wedge$
  $(id_c, j) \in assoc \wedge queue_{sc}(j) = \epsilon \wedge$
  $(mode_s = \texttt{inactive} \vee id_s \neq j)$
Eff: $id_c := \texttt{nil}$
  $q\text{-}stat_{sc}(j) := \texttt{dead}$

reset-nil$_c$
Pre: $\neg(rec_c \vee abrt_c) \wedge$
  $id_c \neq \texttt{nil} \wedge mode_c = \texttt{inactive} \wedge$
  $\forall\, j(id_c, j) \notin assoc \wedge queue_{cs}(id_c) = \epsilon$
Eff: $id_c := \texttt{nil}$
  $q\text{-}stat_{cs}(id_c) := \texttt{dead}$

passive-open
Eff: if $\neg(rec_s \vee abrt_s)$ then
  if $id_s = \texttt{nil}$ then
    $choose\text{-}sid := \texttt{true}$
    $mode_s := \texttt{active}$

choose-server-id(j)
Pre: $choose\text{-}sid = \texttt{true} \wedge j \in SID \setminus used\text{-}id_s$
Eff: $choose\text{-}sid := \texttt{false}$
  $id_s := j$
  $used\text{-}id_s := used\text{-}id_s \cup \{j\}$
  $q\text{-}stat_{sc}(j) := \texttt{live}$

send-msg$_s$(m, close)
Eff: if $\neg(rec_s \vee abrt_s)$ then
  if $mode_s = \texttt{active} \wedge m \neq \texttt{null} \wedge$
  $q\text{-}stat_{sc}(id_s) = \texttt{live}$ then
    $queue_{sc}(id_s) := queue_{sc}(id_s) \cdot m$
  if $close$ then $mode_s := \texttt{inactive}$

receive-msg$_s$(m)
Pre: $\neg(rec_s \vee abrt_s) \wedge$
  $q\text{-}stat_{cs}(i) = \texttt{live} \wedge (i, id_s) \in assoc \wedge$
  $queue_{cs}(i) \neq \epsilon \wedge head(queue_{cs}(i)) = m$
Eff: $queue_{cs}(i) := tail(queue_{cs}(i))$

set-nil$_s$(i)
Pre: $\neg(rec_s \vee abrt_s) \wedge$
  $id_s \neq \texttt{nil} \wedge mode_s = \texttt{inactive} \wedge$
  $(i, id_s) \in assoc \wedge queue_{cs}(i) = \epsilon \wedge$
  $(mode_c = \texttt{inactive} \vee id_c \neq i)$
Eff: $id_s := \texttt{nil}$
  $q\text{-}stat_{cs}(i) := \texttt{dead}$

reset-nil$_s$
Pre: $\neg(rec_s \vee abrt_s) \wedge$
  $id_s \neq \texttt{nil} \wedge mode_s = \texttt{inactive} \wedge$
  $\forall\, i(i, id_s) \notin assoc \wedge queue_{sc}(id_s) = \epsilon$
Eff: $id_s := \texttt{nil}$
  $q\text{-}stat_{sc}(id_s) := \texttt{dead}$

Figure 4-2: *Steps of the specification S. The client side actions are on the left and the server side actions are on the right.*

crash$_c$
Eff: if $id_c \neq \texttt{nil}$ then
$rec_c := \texttt{true}$
if $\exists\, j$ s.t. $(id_c, j) \in assoc$ then
$\forall\, j$ *s.t.*$(id_c, j) \in assoc$,
$queue_{sc}(j) := \epsilon$
$\forall\, j$ *s.t.*$(id_c, j) \in assoc$,
$q\text{-}stat_{sc}(j) := \texttt{dead}$

abort$_c$
Pre: $id_c \neq \texttt{nil}$
Eff: $abrt_c := \texttt{true}$
if $\exists\, j$ s.t. $(id_c, j) \in assoc$ then
$\forall\, j$ *s.t.*$(id_c, j) \in assoc$,
$queue_{sc}(j) := \epsilon$
$\forall\, j$ *s.t.*$(id_c, j) \in assoc$,
$q\text{-}stat_{sc}(j) := \texttt{dead}$

lose$_c$*(I)*
Pre: $(rec_c \vee abrt_c) \wedge$
$I \in$ *suffixes(*$queue_{cs}(id_c)$*)*
Eff: $queue_{cs}(id_c) :=$ delete$(queue_{cs}(id_c), I)$

recover$_c$
Pre: rec_c
Eff: $rec_c := \texttt{false}$
$mode_c := \texttt{inactive}$
if $\forall\, j(id_c, j) \notin assoc \wedge queue_{cs}(id_c) = \epsilon$
then optionally $q\text{-}stat_{cs}(id_c) := \texttt{dead}$
$id_c := \texttt{nil}$

shut-down$_c$
Pre: $abrt_c$
Eff: $abrt_c := \texttt{false}$
$mode_c := \texttt{inactive}$
if $\forall\, j(id_c, j) \notin assoc \wedge queue_{cs}(id_c) = \epsilon$
then optionally $q\text{-}stat_{cs}(id_c) := \texttt{dead}$
$id_c := \texttt{nil}$

crash$_s$
Eff: if $id_s \neq \texttt{nil} \vee mode_s = \texttt{active}$ then
$rec_s := \texttt{true}$
if $\exists\, i$ s.t. $(i, id_s) \in assoc$ then
$\forall\, i$ *s.t.*$(i, id_s) \in assoc$,
$queue_{cs}(i) := \epsilon$
$\forall\, i$ *s.t.*$(i, id_s) \in assoc$,
$q\text{-}stat_{cs}(i) := \texttt{dead}$

abort$_s$
Pre: $id_s \neq \texttt{nil} \vee mode_s = \texttt{active}$
Eff: $abrt_s := \texttt{true}$
if $\exists\, i$ s.t. $(i, id_s) \in assoc$ then
$\forall\, i$ *s.t.*$(i, id_s) \in assoc$,
$queue_{cs}(i) := \epsilon$
$\forall\, i$ *s.t.*$(i, id_s) \in assoc$,
$q\text{-}stat_{cs}(i) := \texttt{dead}$

lose$_s$*(I)*
Pre: $(rec_s \vee abrt_s) \wedge$
$I \in$ *suffixes(*$queue_{sc}(id_s)$*)*
Eff: $queue_{sc}(id_s) :=$ delete$(queue_{sc}(id_s), I)$

recover$_s$
Pre: rec_s
Eff: $rec_s := \texttt{false}$
$mode_s := \texttt{inactive}$
if $\forall\, i(i, id_s) \notin assoc \wedge queue_{sc}(id_s) = \epsilon$
then optionally $q\text{-}stat_{sc}(id_s) := \texttt{dead}$
$id_s := \texttt{nil}$

shut-down$_s$
Pre: $abrt_s$
Eff: $abrt_s := \texttt{false}$
$mode_s := \texttt{inactive}$
if $\forall\, i(i, id_s) \notin assoc \wedge queue_{sc}(id_s) = \epsilon$
then optionally $q\text{-}stat_{sc}(id_s) := \texttt{dead}$
$id_s := \texttt{nil}$

Figure 4-3: *The rest of the steps of the specification S. Client side actions are on the left and server side actions are on the right.*

that the server side user has stopped sending messages.

An association is formed when the internal action *make-assoc(i,j)* is performed. The precondition for the action states that i and j must be in the set of id's used by the client and server respectively, and that neither i nor j has formed any other associations. Under normal conditions, the pair (i, j) is the current value of id_c and id_s. However, when there is a crash and recovery on either side, there may be other types of pairs of id's that can form an association, so we allow the choice to be made nondeterministically. Even though this action is shown on the client side in Figure 4-2, it is not a client or server side action.

The action *receive-msg*$_c$*(m)* is enabled if there is data on $queue_{sc}(j)$ to be passed to the user and an association exists between id_c and j. The precondition guarantees that during any incarnation the client can only receive data from the queue for that incarnation. The action *receive-msg*$_s$*(m)* is symmetric.

The action $set\text{-}nil_c(j)$ sets id_c to `nil` when the client has stopped sending and receiving messages. The condition "mode$_c$ = `inactive`" checks that the client has stopped sending data and "$(mode_s = \texttt{inactive} \vee id_s \neq j) \wedge queue_{sc}(j) = \epsilon$" checks that the server has stopped sending data and that the client has received all the data sent for that connection. The queue indexed by id_c is also flagged as `dead`. The action $set\text{-}nil_s(i)$ is symmetric.

Internal actions $reset\text{-}nil_c$ and $reset\text{-}nil_s$ set id_c and id_s respectively to `nil` if the respective sides receive *open* and *close* signals without receiving any data and before the id's become part of an association pair.

The input action $crash_c$ models crashes on the client side and causes rec_c to be set to `true`, which means the client is in recovery mode, and messages can get lost. We do not want this action to have any effect if the client is closed, hence, the condition in the effect clause. For all queues from which the client can receive data, the $crash_c$ action sets these queues to `dead` and all the messages are deleted. The messages are deleted because after the crash, the client must start a new incarnation, so messages from the previous incarnations are no longer valid. Invariant 4.1, which is defined and proved in the next section, states that there is at most one queue from which the client can receive data in any given state. On the server side $crash_s$ is not quite symmetric because the server can be open and id_s is `nil`. However, if the server is open, either $id_s \neq \texttt{nil}$ or $mode_s = \texttt{active}$.

The $abort_c$ action can be thought of as a client-controlled crash. It allows the client side to immediately abort its side of the connection. That is, it allows the client to close without making sure all the data it sent was delivered or that it received all the data sent to it. This action is necessary because in the low level protocols we want to verify, a host can abort its end of the connection due to conditions that are not based on any external actions. In this action $abrt_c$ is set to `true`, which means the client can lose messages, and also enables the $shut\text{-}down_c$ action. The inclusion of the abort actions in the specification means that the specification does not have the liveness property that barring a crash, messages are eventually delivered. However, in this work we do not verify liveness properties. Furthermore, the low level protocols we study can technically also keep aborting connections without ever delivering any messages. The actions $abort_s$ is symmetric to $abort_c$ except for the precondition.

The action $lose_c(I)$ may occur after a crash or abort and before recovery or shut down on the client side. I is a set of indices from the $queue_{cs}(id_c)$. The *suffixes* of a queue is the set of sets of indices that start from any index j in the queue and includes all the other indices greater than j, that is, sets of consecutive indices at the end of the queue, and *dom(queue)* is the set of indices of *queue*. The function *delete(queue, I)* deletes messages with indices in I from *dom(queue)*. More formally, for any queue q we define *dom(q)*, *suffixes(q)*, and *delete(q,I)* as follows.

$$\begin{aligned} dom(q) &\triangleq \{i \mid 1 \leq i \leq |q|\} \\ suffixes(q) &\triangleq \{\{i \mid j < i \leq |q|\} \mid 0 \leq j \leq |q|\} \\ delete(q,I) &\triangleq \langle q[i] \mid i \in dom(q) \wedge i \notin I\rangle \end{aligned}$$

Elements are deleted form the back of $queue_{cs}(id_c)$ because the server may still receive messages from this queue, but once a message gets lost because of the crash, no message after it may be delivered. On the server side $lose_s(I)$ is symmetric.

The actions $recover_c$ and $recover_s$ signal that the client or server respectively has recovered from a crash, and their effect is to reset variables to values that allow a new connection to be opened. Also if the queue that is indexed by the current id of the host is empty and that id is not part of any association pair, then that queue can no longer be used to send or receive data, so its status is made `dead`.

The actions *shut-down*$_c$ and *shut-down*$_s$ are essentially internal versions of the *recover*$_c$ and *recover*$_s$ actions respectively. They are enabled after the *abort*$_c$ and *abort*$_s$ actions respectively.

Invariants

We define three invariants for the specification S. Recall that an invariant of S is a property that is true of all reachable states of S. We use the standard inductive technique for proving the invariants. That is, we show that the invariants hold for the start states and then show that for every step (u, a, u') of S, if the invariant holds in state u then it also holds in state u'. The state
The first invariant says that each client id may only be be paired with one server id and vice versa.

Invariant 4.1

1. If $(h, j) \in assoc \wedge (i, j) \in assoc$ then $h = i$.

2. If $(i, j) \in assoc \wedge (i, k) \in assoc$ then $j = k$.

Proof: The proof is straightforward, by induction, from the description of the initial values of the variables of S and of *steps(S)*. ■

The next invariant says that all queues that have status `dead` are empty.

Invariant 4.2

1. $\forall\ i \in CID$, if $q\text{-}stat_{cs}(i) =$ `dead` then $queue_{cs}(i) = \epsilon$.

2. $\forall\ j \in SID$, if $q\text{-}stat_{sc}(j) =$ `dead` then $queue_{sc}(j) = \epsilon$.

Proof: The proof is straightforward, by induction, from the description of the initial values of the variables of S and of *steps(S)*. ■

The third invariant says that the number of `live` queues is always finite.

Invariant 4.3

$|live\text{-}q_{cs}|$ and $|live\text{-}q_{sc}|$ are both finite.

Proof: The proof is by induction. We only show the proof for *live*-$q_{cs}(i)$ since the proof for *live*-$q_{sc}(j)$ is symmetric. The base case is the initial state u_0 of S where for all $i \in CID$, $u_0.q\text{-}stat_{cs}(i) = \texttt{dead}$. Thus, $|u_0.live\text{-}q_{cs}| = 0$. For the inductive step we assume the invariant holds for state u_k and show that it holds for state u_{k+1}. By inspection of the steps of S we see that for any step at most one element of $q\text{-}stat_{cs}$ is assigned the flag `live`. Thus, $|u_{k+1}.live\text{-}q_{cs}|$ is bigger than $|u_k.live\text{-}q_{cs}|$ by at most 1, so it is still finite. ■

Invariant I_S is the conjunction of Invariants 4.1, 4.2, and 4.3.

4.2 Delayed-Decision Specification D

In our specification S, messages in the system can be lost, but only if rec_c or rec_s or $abrt_c$ or $abrt_s$ is `true`, that is, we can only lose messages between a crash and a recovery or between an abort and a shut down. In some low-level protocols, whether a message gets lost or not may not be decided until after the host has recovered from the crash or abort. This decision is dependent on *race conditions* that may exist on the channels. For example, in TCP if the client places a message on the channel and then crashes there are several possible scenarios of what could happen to that message. If all copies of the message get dropped by the channel, then it is lost. If it does not get dropped by the channel it may still be lost if the client side recovers and tries to start a new connection. Since the channels are not FIFO, this attempt at the new connection might arrive at the server before the message and cause the server to abort the connection. However, if the message arrives at the server before any other messages it is not lost.

A similar situation comes up in the work of Søgaard-Andersen et al. [35] and they develop the idea of a Delayed-Decision Specification. They then present a backward simulation from the Delayed-Decision Specification to their other specification which is similar to our specification S. We use this idea of a Delayed-Decision Specification, and our specification D is similar to the specification in their work. The need for the backward simulation is suggested by the postponing of nondeterministic choices in the implementations. We show a backward simulation from D to S and then show a refinement mapping from the implementation to D. While we could have done the backward simulation directly from the implementation to S, we use D as an intermediate step because D is very similar to S so the backward simulation from it to S is much simpler than one from the implementation to S would be. Also backward simulations are generally much more complicated than refinement mappings, so our two step simulation turns out to be easier than a direct backward simulation would be.

The Delayed-Decision Specification D looks very much like S, but instead of deleting messages between a crash and a recovery or between an abort and a shut down, D *marks* these messages. Marked messages can then be dropped at any time. Because only marked messages can be dropped, only messages that were in the system at the time of a crash or an abort can be deleted. Marked messages can still be delivered to users.

4.2.1 The automaton D

We define formally the automaton for the specification D. The specification is very similar to that of S with the exception that messages are tagged, *lose* actions are replaced by *drop* actions, and we have the additional internal actions that mark messages.

States and start states

The marks that we put on messages are taken from the following set:

$$Flag \equiv \{\texttt{ok}, \texttt{marked}\}$$

We show only the states that differ from the states of S. All other states are the same and have the same initial values.

Variable	Type	Initially	Description
$queue_{cs}$	an array indexed by CID, of $(Msg \times Flag)^*$	$\forall\ i \in CID$, $queue_{cs}(i) = \epsilon$	An infinite array of FIFO queues. Queues from this array hold messages and their tags sent from the client side. A tag of `ok` means the message cannot get dropped, while a tag of `marked` means the message may get dropped.
$queue_{sc}$	an array indexed by SID, of $(Msg \times Flag)^*$	$\forall\ j \in SID$, $queue_{sc}(j) = \epsilon$	Symmetric to $queue_{cs}$ with messages sent by the server.

We use the normal record notation to extract components of a value or variable. For instance, for an element e of $queue_{cs}(i)$, $e.flag$ and $e.msg$ extract the flag and message respectively from that element. We say an element e of $queue_{cs}(i)$ or $queue_{sc}(j)$ is `marked` if $e.flag$ = `marked`.

The derived variables $live\text{-}q_{cs}$ and $live\text{-}q_{sc}$ are defined for D exactly as they are defined for S. We also have an additional derived variable for D. Let q_D be a queue in the set $(Msg \times Flag)^*$, that is, q_D has the same type as queues in D. Then define $\#\mathtt{ok}(q_D)$ to be the number of elements e of q_D with $e.flag$ = `ok`. These derived variables are used in showing the simulation from D to S.

Action signature of D

The user interface of D is the same as that of S. D has the additional internal actions $mark_c(I)$, $mark_s(I)$, $drop_c(I, k)$, and $drop_s(I, l)$, and does not have the $lose_c(I)$ and $lose_s(I)$ actions.

Steps

The steps of D that are not in S or are different are shown in Figure 4-4. The step rule for $mark_c(I)$ and $mark_s(I)$ uses a function $mark$ which is intended to mark messages with indices in I. Formally, for any queue $q \in (\mathrm{Msg} \times \mathrm{Flag})^*$ and any set $I \subseteq dom(q)$, define:

$$mark(q, I) \triangleq \langle (\text{if } i \in I \text{ then } (q[i].msg, \mathtt{marked}) \text{ else } q[i]) \mid i \in dom(q) \rangle$$

The steps for D are mostly the same as the steps for S expect for a few changes and additions. The first change is that the messages are tagged, so when messages come in from the users they are tagged with `ok`, and before they get delivered the tags are removed. Note that even messages tagged with `marked` can be delivered. The other change is that instead of having $lose_c(I)$, $lose_s(I)$ actions that may delete messages between a crash and a recovery or an abort and a shut down, there are now $mark_c(I)$, $drop_c(I, k)$, $mark_s(I)$, and $drop_s(I, l)$ actions. Messages can get `marked` only between a crash and recovery or between an abort and shut down, but can be dropped at anytime. The parameters k and l

send-msg$_c$*(open, m, close)*
Eff: if $\neg(rec_c \vee abrt_c)$ then
  if $open \wedge id_c =$ `nil` then
    $id_c :\in CID \setminus used\text{-}id_c$
    $used\text{-}id_c := used\text{-}id_c \cup \{id_c\}$
    $mode_c :=$ `active`
    $q\text{-}stat_{cs}(id_c) :=$ `live`
  if $mode_c =$ `active`$\wedge\ m \neq$ `null` $\wedge$
  $q\text{-}stat_{cs}(id_c) =$ `live` then
    $queue_{cs}(id_c) := queue_{cs}(id_c) \cdot (m, \mathtt{ok})$
  if $close$ then $mode_c :=$ `inactive`

receive-msg$_c$*(m)*
Pre: $\neg(rec_c \vee abrt_c) \wedge$
  $queue_{sc}(j) \neq \epsilon \wedge$
  $q\text{-}stat_{sc}(j) =$ `live` $\wedge\ (id_c, \mathrm{j}) \in assoc \wedge$
  $(\mathrm{head}(queue_{sc}(j))).\mathrm{msg} = m$
Eff: $queue_{sc}(j) := \mathrm{tail}(queue_{sc}(j))$

$mark_c(I)$
Pre: $(rec_c \vee abrt_c) \wedge$
  $I \in suffixes(queue_{cs}(id_c))$
Eff: $queue_{cs}(id_c) := mark(queue_{cs}(id_c), I)$

$drop_c(I, k)$
Pre: $queue_{cs}(k) =$ `live` $\wedge$
  $I \in suffixes(queue_{cs}(k)) \wedge$
  $\forall\ i \in I\ queue_{cs}(k)[i].\mathrm{flag} =$ `marked`
Eff: $queue_{cs}(k) := \mathrm{delete}(queue_{cs}(k), I)$

send-msg$_s$*(m, close)*
Eff: if $\neg(rec_s \vee abrt_s)$ then
  if $mode_s =$ `active` $\wedge\ m \neq$ `null` $\wedge$
  $q\text{-}stat_{sc}(id_s) =$ `live` then
    $queue_{sc}(id_s) := queue_{sc}(id_s) \cdot (m, \mathtt{ok})$
  if $close$ then $mode_s :=$ `inactive`

receive-msg$_s$*(m)*
Pre: $\neg(rec_s \vee abrt_s) \wedge$
  $queue_{cs}(i) \neq \epsilon \wedge$
  $q\text{-}stat_{cs}(i) =$ `live` $\wedge\ (i, id_s) \in assoc \wedge$
  $(\mathrm{head}(queue_{cs}(i))).\mathrm{msg} = m$
Eff: $queue_{cs}(i) := \mathrm{tail}(queue_{cs}(i))$

$mark_s(I)$
Pre: $(rec_s \vee abrt_s) \wedge$
  $I \in suffixes(queue_{sc}(id_s))$
Eff: $queue_{sc}(id_s) := mark(queue_{sc}(id_s), I)$

$drop_s(I, l)$
Pre: $queue_{sc}(l) =$ `live` $\wedge$
  $I \in suffixes(queue_{sc}(l)) \wedge$
  $\forall\ i \in I\ queue_{sc}(l)[i].\mathrm{flag} =$ `marked`
Eff: $queue_{sc}(l) := \mathrm{delete}(queue_{sc}(l), I)$

Figure 4-4: *Steps of the Delayed-Decision Specification D that differ from the steps of S.*

in the drop actions allow nondeterminism in the queues from which messages get dropped. Because messages can only be marked between a crash and a recovery or an abort and a shut down, only messages from the queue indexed by the current id of the client or server can be marked. However, because marked messages can be deleted anytime, the queue from which the message is deleted may not be the queue indexed by the current id. Furthermore, there may be several `live` queues with marked messages. Therefore, the extra parameters in the $drop_c(I, k)$ and $drop_s(I, l)$ actions allow nondeterministic choice of the queue from which to drop the message.

4.2.2 The correctness of D

In this section we prove the correctness of D with respect to S. In this work we use s to refer to states of the low level protocol and u to refer to steps of the higher level abstract specification. For this proof s refers to states of D and u to states of S.

Invariants

Invariants 4.1, 4.2, and 4.3 which we proved for the states of S in Section 4.1.2 also hold for the states of D. That is, the properties stated below are true for all reachable states of D.

Invariant 4.4

1. If $(h, j) \in assoc \wedge (i, j) \in assoc$ then $h = i$.

2. If $(i, j) \in assoc \wedge (i, k) \in assoc$ then $j = k$.

Proof: The proof is straightforward, by induction, from the description of the initial values of the variables of S and of *steps(D)*. ■

Invariant 4.5

1. $\forall\ i \in CID$, if $q\text{-}stat_{cs}(i) = \texttt{dead}$ then $queue_{cs}(i) = \epsilon$.

2. $\forall\ j \in SID$, if $q\text{-}stat_{sc}(j) = \texttt{dead}$ then $queue_{sc}(j) = \epsilon$.

Proof: The proof is straightforward, by induction, from the description of the initial values of the variables of D and of *steps(D)*. ■

Invariant 4.6

$|live\text{-}q_c|$ and $|live\text{-}q_s|$ are both finite.

Proof: The proof is the same as the proof for Invariant 4.3. ■

The Invariant I_D is the conjunction of Invariants 4.4, 4.5 and 4.6.

The simulation

We prove the correctness of D by showing an image finite backward simulation from D to S. The proof is very similar to the one given in [35], and most of the definitions and lemmas are the same. The main differences are that we have multiple queues going in both directions in both S and D instead of the single queues for each in [35], and we do not have the *status* variable which is a significant part of the complexity of the simulation in [35]. Before we show the backward simulation we need a few preliminary definitions and lemmas. Let q_S be in the set Msg^*, that is, q_S has the same type as queues in S.

Definition 4.1 (Explanation)

Define an *explanation* from q_S to q_D to be any mapping $f : dom(q_S) \rightarrow dom(q_D)$ that satisfies the following four conditions:

1. f is total
2. f is strictly increasing
3. $\forall\ i\ \in\ dom(q_D) \setminus rng(f) : q_D[i].flag = \texttt{marked}$
4. $\forall\ i\ \in\ dom(q_S) : q_D[f(i)].msg = q_S[i]$ □

Intuitively this means if there exists an explanation from q_S to q_D, then q_S can be obtained from q_D by first deleting some of the marked elements of q_D and then removing the flags from the remaining elements.

Lemma 4.1

If f is an explanation from q_S to q_D, then $|q_S| \leq |q_D|$.

Proof: Suppose $|q_S| > |q_D|$. Then it is impossible to find a mapping from $dom(q_S)$ to $dom(q_D)$ that is total and strictly increasing, thus conditions 1 and 2 are violated, hence $|q_S| \leq |q_D|$. ■

Lemma 4.2

If f is an explanation from q_S to q_D, then $|q_S| \geq \#\texttt{ok}(q_D)$.

Proof: Suppose $|q_S| < \#\texttt{ok}(q_D)$. Then conditions 1 and 2 of Definition 4.1 give us that $|rng(f)| = |q_S| < \#\texttt{ok}(q_D)$, so there must exist indices i in q_D such $q_D[i]$.flag = $\texttt{ok}$ and $i \notin rng(f)$. However, this contradicts condition 3 of Definition 4.1, and therefore we can conclude that $|q_S| \geq \#\texttt{ok}(q_D)$. ■

We are now ready to define B_{DS} over $states(D) \times states(S)$. See [35] for a discussion and some intuition on how to define backward simulations in general.

Definition 4.2 (Image-Finite Backward Simulation from D to S)
If $s \in$ *states(D)* and $u \in$ *states(S)*, then define that $(s, u) \in B_{\text{DS}}$ if the following conditions hold:

1. $u.assoc = s.assoc$
2. $u.choose\text{-}sid = s.choose\text{-}sid$
3. $u.used\text{-}id_c = s.used\text{-}id_c$
   $u.used\text{-}id_s = s.used\text{-}id_s$
4. $u.rec_c = s.rec_c$
   $u.rec_s = s.rec_s$
5. $u.abrt_c = s.abrt_c$
   $u.abrt_s = s.abrt_s$
6. $u.id_c = s.id_c$
   $u.id_s = s.id_s$
7. $u.mode_c = s.mode_c$
   $u.mode_s = s.mode_s$
8. $\forall\ i \in CID\ u.q\text{-}stat_{cs}(i) = s.q\text{-}stat_{cs}(i)$
   $\forall\ j \in SID\ u.q\text{-}stat_{sc}(j) = s.q\text{-}stat_{sc}(j)$
9. $(\forall\ i \in CID)$ ($\exists$ explanation f_i from $u.queue_{cs}(i)$ to $s.queue_{cs}(i)$)
   $(\forall\ j \in SID)$ ($\exists$ explanation g_i from $u.queue_{sc}(j)$ to $s.queue_{sc}(j)$)

Each of the variables in S other than the queues is equal to its counterpart in D. In the proof below when we write u.variables = s.variables we mean the eight sets of equations of items one through eight in Definition 4.2.

We now state some preliminary lemmas that simplify the main proof. Let *maxqueue* be a function of type: *(Msg* $\times$ *Flag)** $\rightarrow$ *Msg** such that for any q_D, *maxqueue*(q_D) is defined to be the queue q_S obtained by removing all flag components from q_D. Formally, we have

$$q_S = maxqueue(q_D) \text{ iff } |q_S| = |q_D| \text{ and } \forall i \in dom(q_D) : q_S[i] = q_D[i].msg.$$

Lemma 4.3
The identity mapping f from dom(q_D) *is an explanation from maxqueue(q_D) to q_D.*

Proof: Conditions 1 and 2 of Definition 4.1 are satisfied since the identity mapping is both total and strictly increasing. Condition 3 is also satisfied since $rng(f) = dom(q_D)$. Finally from the definition of *maxqueue* we directly see that condition 4 is also satisfied. ∎

Lemma 4.4

Let $s \in$ states(D). Then there exists a state $u \in$ states(S) such that $(s, u) \in B_{\text{DS}}$.

Proof: Let $q_{S_i} = maxqueue(s.queue_{cs}(i))\ \forall\ i \in CID$, and $q^1_{S_j} = maxqueue(s.queue_{sc}(j))$ $\forall\ j \in SID$. Then by Lemma 4.3 there exists an explanation from q_{S_i} to $s.queue_{cs}(i)$ and an explanation from $q^1_{S_j}$ to $s.queue_{sc}(j)$. Thus, if we have $u.queue_{cs}(i) = q_{S_i}$, $u.queue_{sc}(j)$ $= q^1_{S_j}$, and for all the other variables u.variables = s.variables, this gives a state u such that $(s, u) \in B_{\text{DS}}$. ∎

Lemma 4.5

$D \leq_{iB} S$ via B_{DS} with respect to I_D and I_S.

Proof: We first show that B_{DS} is image-finite and then check the three conditions of Definition 3.10 which we call non-emptiness, base case, and inductive case respectively.

Let s be an arbitrary state of D. We must show that there exists only finitely many states u of S such that $(s, u) \in B_{\text{DS}}$. All the variables in s except for the queues are equal to their counterparts in u, so these variables cannot cause infinitely many states. It now remains to be shown that for a fixed but arbitrary s, that $\forall\ i \in CID$ and $\forall\ j \in SID$, $s.queue_{cs}(i)$ and $s.queue_{sc}(j)$ can only take on finitely many values. We only show this for $s.queue_{cs}(i)$ as the proof for $s.queue_{sc}(j)$ is symmetric. From Invariant 4.5 we know that if $s.q\text{-}stat_{cs}(i)$ is `dead`, then $s.queue_{cs}(i)$ is empty. Thus, even though there are infinitely many of these queues, since they all have only one possible value, these queues do not cause infinitely many states. From Invariant 4.6 we know that there are finitely many $s.queue_{cs}(i)$ such that $s.q\text{-}stat_{cs}(i)$ is `live`. For each such queue, Lemma 4.1 gives us that $|u.queue_{cs}(i)| \leq |s.queue_{cs}(i)|$, thus, there are only a finite number of lengths to choose from ($s.queue_{cs}(i)$ is finite). Also, there exists only a finite number of mappings (explanations) between two finite domains. Finally, condition 4 of Definition 4.1 gives us that values of the elements of the possible $u.queue_{cs}(i)$ are uniquely determined by $s.queue_{cs}(i)$ and the (finitely many) explanations. Hence, each $u.queue_{cs}(i)$ can only take on finitely many values given s, and since there are only a finite number of these queues, there are only finitely many states u. ∎

Non-emptiness

Non-emptiness follows immediately from Lemma 4.4

Base Case

Let s_0 be the (unique) start state of D. Then if $(s_0, u) \in B_{\mathrm{DS}}$, then u.variables $= s$.variables and $u.queue_{cs}(i) = u.queue_{sc}(j) = \epsilon$. Thus, u is the unique start state of S.

Inductive Case

Assume $(s, a, s') \in$ *steps(D)* and let u' be an arbitrary state of S such that $(s', u') \in B_{\mathrm{DS}}$. Below we consider cases based on a and for each case we define a finite execution fragment α of S with $lstate(\alpha) = u'$, $(s, fstate(\alpha)) \in B_{\mathrm{DS}}$, and $trace(\alpha) = trace(a)$. In order to show $(s, fstate(\alpha)) \in B_{\mathrm{DS}}$, we need to show that the value of the state variables for state s and $fstate(\alpha) = u$ are related according to our definition of B_{DS}. In most of the cases below α $= (u, a, u')$, and for these cases it is trivially true that for a state u such that $(s, u) \in B_{\mathrm{DS}}$, u.variables $= s$.variables. The interesting aspect of showing $(s, u) \in B_{\mathrm{DS}}$ is showing that we can find valid explanations from the queues in state u to the queues in state s.

a = *send-msg*$_c$*(open, m, close)*.

In this case we show that we can define u such that *(u, send-msg*$_c$*(open, m, close), u')* $\in$ *steps(S)* and $(s, u) \in B_{\mathrm{DS}}$. Clearly the step has the right trace. This step has eight permutations depending on whether *open* and *close* are true or false and whether m is `null` or not. The eight permutations gives eight subcases, but the only non-trivial subcases are the ones where $m \neq$ `null`, and in those subcases where $s'.queue_{cs}(s'.id_c) = s.queue_{cs}(s.id_c)$ $\cdot(m,$ `ok`$)$. For all other cases, the queues do not change, so the explanations that we know exist because $(s', u') \in B_{\mathrm{DS}}$ are also explanations for (s, u), and u.variables $= s$.variables. In all the non-trivial subcases we also have u.variables $= s$.variables, and the same construction of the explanations works for all of them. We show this construction below.

Define $u.queue_{cs}(u.id_c) = init(u'.queue_{cs}(u'.id_c))$. We only need to find an explanation from $u.queue_{cs}(u.id_c)$ to $s.queue_{cs}(s.id_c)$. (Since this action does not change $u.queue_{sc}(j)$ nor $s.queue_{sc}(j)$ $\forall\ j \in SID$ nor $u.queue_{cs}(i)$ nor $s.queue_{cs}(i)$ $\forall i \in CID \wedge i \neq s.id_c$, the explanations that exist between these queues in states s' and u' are also explanations between the same queues in states s and u.) Let f'_{id_c} be an explanation from $u' queue_{cs}(u'.id_c)$ to $s'.queue_{cs}(s'.id_c)$. Such an explanation exists since $(s', u') \in B_{\mathrm{DS}}$. Since $last(s'.queue_{cs}(s'.id_c)).flag =$ `ok`, we have from Lemma 4.2 and the definition of an explanation that $f'_{id_c}(maxidx(u'.queue_{cs}(u'.id_c))) = maxidx(s'.queue_{cs}(s'.id_c))$. Then

$$f_{id_c} = f'_{id_c} \restriction dom(u.queue_{cs}(u.id_c))$$

is clearly an explanation from $u.queue_{cs}(u.id_c)$ to $s.queue_{cs}(s.id_c)$.

<u>a = *passive-open, choose-server-id(j)*</u>.

For this case it is easy to see we can define u such that *(u, passive-open, u')* and *(u, choose-server-id(j), u')* respectively $\in$ *steps(S)* and $(s, u) \in B_{\mathrm{DS}}$. Such a state u has u.variables = s.variables, and since the actions do not affect the elements of any queues, all the explanations for $(s', u') \in B_{\mathrm{DS}}$ are explanations for (s, u).

<u>a = *send-msg$_s$(m, close)*</u>.

For this case α = *(u, send-msg$_s$, u')*. This action has four permutations and is similar to the case for *send-msg$_c$(open, m, close)*. Like that case, the non-trivial subcases occur when $s.queue_{sc}(s.id_s)$ changes as a result of a and $u.queue_{sc}(u.id_s)$ changes as result of α. We can find an explanation

$$g_{id_s} = g'_{id_s} \restriction dom(u.queue_{sc}(u.id_s))$$

from $u.queue_{sc}(u.id_s)$ to $s.queue_{sc}(s.id_s)$, where g'_{id_s} is the explanation we know exists from $u'queue_{sc}(u'.id_s)$ to $s'.queue_{sc}(s'.id_s)$.

<u>a = *make-assoc(i,j)*</u>.

This is another case where the action does not affect the queues, so it is easy to define u such that α = *(u, make-assoc(i,j), u')* $\in$ *steps(S)* and $(s, u) \in B_{\mathrm{DS}}$. Clearly the step has the right trace (the empty trace). We let u.variables = s.variables and the explanations that work from state u' to s' works from state u to state s.

<u>a = *receive-msg$_c$(m)*</u>.

For this case α = *(u, receive-msg$_c$(m), u')*. We need to show that there is a state u such that $(s, u) \in B_{\mathrm{DS}}$. Let u.variables = s.variables. We only need to show the explanation from $u.queue_{sc}(j)$ to $s.queue_{sc}(j)$ since all other queues are unaffected by the action. Let f'_j be an explanation from $u'.queue_{sc}(j)$ to $s'.queue_{sc}(j)$. Then we can define f_j in the following way.

$$f_j = [(i+1) \mapsto (f'_j(i)+1) | i \in dom(f'_j)] \cup [1 \mapsto 1]$$

Intuitively f_j relates the same elements in $u.queue_{sc}(j)$ and $s.queue_{sc}(j)$ that were related by f'_j in $u'.queue_{sc}(j)$ and $s'.queue_{sc}(j)$ (these elements all have their indices increased by one because of the new elements at the head of the queues), and relates the messages m ($[1 \mapsto 1]$). It is easy to see that f_j is an explanation.

$a = \textit{receive-msg}_s(m)$.

This case is symmetric to the case for $\textit{receive-msg}_c(m)$.

$a = \textit{set-nil}_c(j),\ \textit{reset-nil}_c,\ \textit{recover}_c,\ \textit{shut-down}_c$.

For all theses cases $\alpha = (u, a, u')$. In the state u such that $(u, s) \in B_{\text{DS}}$, u.variables $=$ s.variables and the explanations from queues of u' to queues of s' are valid from u to s since the actions do not affect the contents of any queues.

$a = \textit{set-nil}_s(i),\ \textit{reset-nil}_s,\ \textit{recover}_s,\ \textit{shut-down}_s$.

These cases are symmetric to the cases for $\textit{set-nil}_c(j)$, $\textit{reset-nil}_c$, $\textit{recover}_c$, and $\textit{shut-down}_c$ respectively.

$a = \textit{crash}_c$.

We can define u such $(u,\ \textit{crash}_c,\ u') \in \textit{steps}(S)$ and $(s, u) \in B_{\text{DS}}$. For this step u.variables $=$ s.variables and clearly the explanations from $u'.queue_{cs}(i)$ to $s'.queue_{cs}(i)$ are also explanations from $u.queue_{cs}(i)$ to $s.queue_{cs}(i)$ $\forall\ i \in CID$, and also the explanations from $u'.queue_{sc}(k)$ to $s'.queue_{sc}(k)$ are explanations for $u.queue_{sc}(k)$ to $s.queue_{sc}(k)$ $\forall\ k \in SID \wedge k \neq j$ since the action does not affect these queues. We now define $u.queue_{sc}(j)$ and show that an explanation exists to $s.queue_{sc}(j)$. Let $u.queue_{sc}(j) = maxqueue(s.queue_{sc}(j))$, then by Lemma 4.3 the identity mapping from $dom(u.queue_{sc}(j))$ to $dom(s.queue_{sc}(j))$ is an explanation.

$a = \textit{crash}_s$.

This case is symmetric to the case for $\textit{crash}_c$.

$a = \textit{abort}_c$.

For this case we define u such that $(u,\ \textit{abort}_c,\ u') \in \textit{steps}(S)$ and $(s, u) \in B_{\text{DS}}$. Clearly the traces are the same. For this step u.variables = s.variables, and since $\forall\ i\ \in CID$ the elements of $s.queue_{cs}(i)$ and $u.queue_{cs}(i)$ are not affected by $abort_c$, the explanations between these queues which we know exist are explanations in states s and u. Also the explanations from $u'.queue_{sc}(k)$ to $s'.queue_{sc}(k)$ are explanations for $u.queue_{sc}(k)$ to $s.queue_{sc}(k)\ \forall\ k\ \in SID\ \wedge\ k \neq\ j$ since the actions do not affect the elements of these queues. Therefore, we only need to show explanations from $u.queue_{sc}(j)$ to $s.queue_{sc}(j)$. Let $u.queue_{sc}(j) = maxqueue(s.queue_{sc}(j))$, then by Lemma 4.3, the identity mapping from $dom(u.queue_{sc}(j))$ to $dom(s.queue_{sc}(j))$ is an explanation.

$\underline{a\ =\ abort_s}$.

This case is symmetric to the case for $abort_c$.

$\underline{a\ =\ mark_c(I)}$.

In this case we can define u and I' such that $(u,\ lose_c(I'),\ u') \in steps(S)$ and $(s, u) \in B_{\mathrm{DS}}$. Clearly $trace(\alpha) = trace(a)$. Since $\forall\ j\ \in SID\ s.queue_{sc}(j)$ is not affected by $mark_c(I)$ and $u.queue_{sc}(j)$ is not affected by $lose_c(I')$; and $\forall\ i\ \in CID\ \wedge\ i \neq s.id_c$, $s.queue_{cs}(i)$ is not affected by $mark_c(I)$ and $u.queue_{cs}(i)$ is not affected by $lose_c(I')$, the explanations between these queues which we know exist are explanation in states s and u also. Therefore, we only need to construct an explanation from $u.queue_{cs}(u.id_c)$ to $s.queue_{cs}(s.id_c)$. Let u.variables = s.variables and $u.queue_{cs}(u.id_c) = maxqueue(s.queue_{cs}(s.\mathrm{id}_c))$; then by Lemma 4.3, the identity mapping is an explanation from $u.queue_{cs}(u.id_c)$ to $s.queue_{cs}(s.id_c)$.

We now need to show that $lose_c(I')$ is enabled from state u in S. Since u.variables = s.variables and $mark_c(I)$ is enabled in s, we know $s.rec_c\ =\ u.rec_c\ =$ `true`, and that $I \in suffixes(s.queue_{cs}(s.id_c)$. To define an appropriate I' we first observe that $maxqueue(s.queue_{cs}(s.id_c)) = maxqueue(s'.queue_{cs}(s'.id_c))$, and since $u.queue_{cs}(u.id_c) = maxqueue(s.queue_{cs}(s.id_c))$, it is easy to see we can obtain $u'.queue_{cs}(u'.id_c)$ from $u.queue_{cs}(u.id_c)$ by deleting some (possibly zero) elements that are in $suffixes(u.queue_{cs}(u.id_c))$. Thus, I is an appropriate I', that is, $I' = I$.

$\underline{a\ =\ mark_s(I)}$.

This action is symmetric to the previous case.

$\underline{a\ =\ drop_c(I, k)}$.

The corresponding action in S is the empty step, i.e., $(s, u') \in B_{\mathrm{DS}}$. Since $drop_c(I, k)$ is internal the empty step has the right trace. This action only affects $s.queue_{cs}(k)$, so we only need to an explanation from $u.queue_{cs}(k)$ to $s.queue_{cs}(k)$. Let f'_k be an arbitrary explanation form $u'.queue_{cs}(k)$ to $s'.queue_{cs}(k)$ (we know one exists because $(s', u') \in B_{\mathrm{DS}}$). I contains the indices of the elements of $s.queue_{cs}(k)$ that were deleted in the $drop_c(I, k)$ step. Then $|dom(s'.queue_{cs}(k))| = |dom(s.queue_{cs}(k)) \setminus I|$. Now let h be the unique bijective, strictly increasing mapping from $dom(s'.queue_{cs}(k))$ to $dom(s.queue_{cs}(k)) \setminus I$. Informally h maps indices of elements in $s'.queue_{cs}(k)$ to the indices the same elements had in $s.queue_{cs}(k)$. Define $f_k = h \circ f'_k$. To Check that f_k is a valid explanation from $u.queue_{cs}(k)$ to $s.queue_{cs}(k)$, we check conditions 1-4 of Definition 4.1.

`Conditions 1 and 2`

Since f'_k is total and strictly increasing from $dom(u'.queue_{cs}(k))$ to $dom(s'.queue_{cs}(k))$ and h is total and strictly increasing from $dom(s'.queue_{cs}(k))$ to $dom(s.queue_{cs}(k)) \setminus I$, f_k is total and strictly increasing from $dom(u.queue_{cs}(k))$ to $dom(s.queue_{cs}(k))$.

`Condition 3`

We have that the $dom(s.queue_{cs}(k)) \setminus rng(h \circ f'_k) = I \cup h(dom(s'.queue_{cs}(k)) \setminus rng(f'_k))$. Informally, this means if an element in $s.queue_{cs}(k)$ is not "hit" by f_k, then this is because it was either one of the elements that are deleted in $drop_c(I, k)$ or it was not "hit" by f'_k. We know that the elements in I are `marked` because that is a precondition for them to be dropped. Furthermore, since f'_k is an explanation we know that the elements in $s'.queue_{cs}(k)$ not hit by f'_k are `marked`. Now since h maps indices of elements in $s'.queue_{cs}(k)$ to the indices the same elements had in $s.queue_{cs}(k)$, we know that these elements in $s.queue_{cs}(k)$ are also `marked`.

`Condition 4`

By the fact that f'_k is an explanation (and therefore satisfies condition 4) and the fact that h maps the index of an element to the index of the same element, it directly follows that f_k satisfies condition 4.

$\underline{a = drop_s(I, l)}$.

This action is symmetric to the previous case.

This concludes the backward simulation proof. ■

Theorem 4.1

The traces of D are a subset of the traces of S, that is, $D \sqsubseteq S$.

Proof: The proof follows directly from Lemma 4.5 and the soundness of backward simulations (Theorem 3.4). ∎

Chapter 5

The Communication Channels

In this chapter we present the formal definitions of the communication channels used by TCP and T/TCP with stable and unbounded counters and TCP with bounded counters. The formal definition is meant to be an abstract model of the service provided by the IP layer of the TCP/IP Internet layering model. We use the term "communication channels" because the service of IP we are interested in modeling is the same as what is generally referred to as unreliable channels in the literature [35, 21]. That is, IP receives packets from a sender and if a packet is not dropped, it is eventually delivered to a receiver. Also IP does not create any spurious packets nor does it corrupt packets it receives. However, packets can be lost, duplicated and reordered.

In order for TCP with bounded counters to work correctly, a maximum segment lifetime (MSL) must be imposed on packets placed on the channels. That is, a packet that is placed on a channel and does not get delivered within the MSL gets dropped from the channel. We represent the value of the MSL as μ. Therefore, the automata for these channels are represented as GTA. For TCP and T/TCP with unbounded counters, this property of the channel is not required for correct behavior, so the channels for these protocols are represented as untimed automata.

In Chapter 11 where we present the impossibility result, we define a slightly different model for channels, that includes liveness properties that we do not needed for the verification of the protocols.

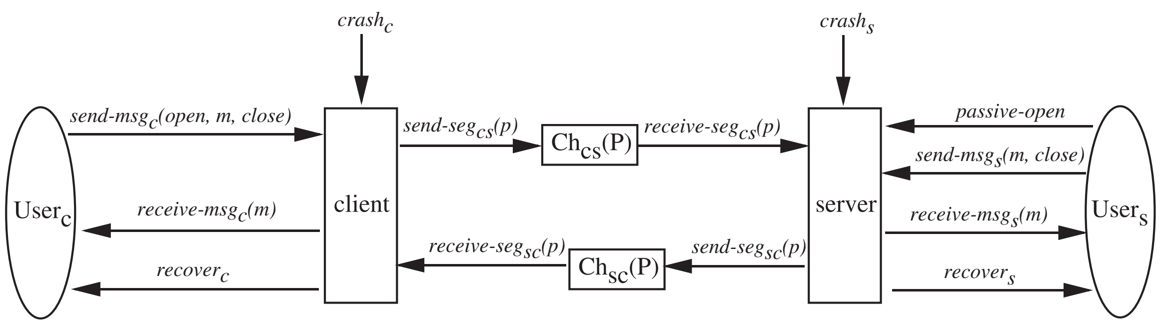

Figure 5-1: *Communication channels link the client and server.*

5.1 The untimed channel automaton

In this section we describe the formal model for the channels used by TCP and T/TCP with unbounded counters. These channels are presented as untimed automata. The channels are parameterized with a set of possible packets $\mathcal{P}$. Figure 5-1 shows the two channels we use and their user interface. Channel $Ch_{cs}(\mathcal{P})$ is for packets from the client to the server and $Ch_{sc}(\mathcal{P})$ is for packets in the other direction. We only specify $Ch_{cs}(\mathcal{P})$ since $Ch_{cs}(\mathcal{P})$ is exactly symmetric.

5.1.1 States and Start States

As mentioned above, $Ch_{cs}(\mathcal{P})$ is parameterized with a set of possible packets $\mathcal{P}$. The channel automaton only only has one state variable $\mathit{in\text{-}transit}_{cs}$ which is a multiset of the packets (including duplicates) currently in the channel. We use the notation $\mathcal{B}(S)$ to denote a multiset or bag of all (finite or infinite) bags with elements from a set S.

Variable	**Type**	**Initially**	**Description**
$\mathit{in\text{-}transit}_{cs}$	$\mathcal{B}(P)$	$\emptyset$	A multiset of packets.

Action Signature

Input:
$\mathit{send\text{-}seg}_{cs}(p)$, $p \in \mathcal{P}$

Output:
$\mathit{receive\text{-}seg}_{cs}(p)$, $p \in \mathcal{P}$

Internal:
$\mathit{drop}_{cs}(p)$, $p \in \mathcal{P}$
$\mathit{duplicate}_{cs}(p)$, $p \in \mathcal{P}$

$send\text{-}seg_{cs}(p)$ `Eff:` $in\text{-}transit_{cs} := in\text{-}transit_{cs} \cup \{p\}$	$receive\text{-}seg_{cs}(p)$ `Pre:` $(p) \in in\text{-}transit_{cs}$ `Eff:` $in\text{-}transit_{cs} := in\text{-}transit_{cs} \setminus \{p\}$
$drop_{cs}(p)$ `Pre:` $(p) \in in\text{-}transit_{cs}$ `Eff:` $in\text{-}transit_{cs} := in\text{-}transit_{cs} \setminus \{p\}$	$duplicate_{cs}(p)$ `Pre:` $(p) \in in\text{-}transit_{cs}$ `Eff:` $in\text{-}transit_{cs} := in\text{-}transit_{cs} \cup \{p\}$

Figure 5-2: *The steps of* $Ch_{cs}(\mathcal{P})$.

5.1.2 Steps

The steps of $Ch_{cs}(\mathcal{P})$ are straightforward and are shown in Figure 5-2. The $send\text{-}seg_{cs}(p)$ input action comes from the external user of the channel, which in this case is the client of the TCP or T/TCP host with stable and unbounded counters. The effect of this action is to place the packet p in the multiset $in\text{-}transit_{cs}$. The complimentary action, $receive\text{-}seg_{cs}(p)$, removes a single element from the multiset and passes the packet p to the user (the server TCP or T/TCP host). The internal actions $duplicate_{cs}(p)$ and $drop_{cs}(p)$ duplicate and remove elements of $in\text{-}transit_{cs}$ respectively.

5.2 The channel GTA

In this section we present the GTA for the channel automaton for TCP with bounded counters. Again we only specify the automaton for packets for the client to the server. The specification is very similar to the specification for $Ch_{cs}(\mathcal{P})$. Now when a packet gets placed on the channel, it is paired with the current time (its *send time*) and it must get dropped when the difference between the current time and its *send time* is equal to μ, and it may get dropped even if its time on the channel has not exceeded μ.

5.2.1 States and Start States

Because of the maximum segment lifetime we refer to the channel for packets from the client to the server as $\mu Ch_{cs}(\mathcal{P})$ and the channel for packets from the server to the client as $\mu Ch_{sc}(\mathcal{P})$. The channel $\mu Ch_{cs}(\mathcal{P})$ is also parameterized with a set of possible packets $\mathcal{P}$. The type `T` ranges over the nonnegative real numbers and represents time. In addition to *now* that represents real time, the other state variable is $in\text{-}transit_{cs}$ which is a multiset of

send-seg$_{cs}(p)$
`Eff:` $\textit{in-transit}_{cs} := \textit{in-transit}_{cs} \cup \{(p, (now_{cs} + \mu)\}$

drop$_{cs}$*(p,t)*
`Pre:` $(p, t) \in \textit{in-transit}_{cs}$
`Eff:` $\textit{in-transit}_{cs} := \textit{in-transit}_{cs} \setminus \{(p, t)\}$

duplicate$_{cs}$*(p,t)*
`Pre:` $(p, t) \in \textit{in-transit}_{cs}$
`Eff:` $\textit{in-transit}_{cs} := \textit{in-transit}_{cs} \cup \{(p, t)\}$

receive-seg$_{cs}(p)$
`Pre:` $(p, t) \in \textit{in-transit}_{cs}$
`Eff:` $\textit{in-transit}_{cs} := \textit{in-transit}_{cs} \setminus \{(p, t)\}$

$\nu(t)$ (time-passage)
`Pre:` $\forall (p, t') \in \textit{in-transit}_{cs} : (now \;\; + t \leq t')$
`Eff:` $now_{cs} := now_{cs} + t$

Figure 5-3: *The steps of* $\mu Ch_{cs}(\mathcal{P})$.

the packets (including duplicates) currently in the channel, paired with their *send time*.

Variable	**Type**	**Initially**	**Description**
now_{cs}	`T`	0	Real time.
$\textit{in-transit}_{cs}$	$\mathcal{B}(P \times \mathtt{T})$	$\emptyset$	A multiset of packets together with the time the packets were sent.

Action Signature

Input:
  send-seg$_{cs}(p)$, $p \in \mathcal{P}$

Output:
  receive-seg$_{cs}(p)$, $p \in \mathcal{P}$

Internal:
  drop$_{cs}(p, t)$, $p \in \mathcal{P}$ and $t \in \mathtt{T}$
  duplicate$_{cs}(p, t)$, $p \in \mathcal{P}$ and $t \in \mathtt{T}$

Time-passage:
  $\nu(t), t \in R^{+}$

5.2.2 Steps

The steps of $\mu Ch_{cs}(\mathcal{P})$ are are shown in Figure 5-3. The *send-seg*$_{cs}(p)$ input action comes from the external user of the channel, which in this case is the client host for TCP with bounded counters. The effect of this action is to place the packet p paired with a time-stamp (now) in the multiset *in-transit*$_{cs}$. The complimentary action, *receive-seg*$_{cs}(p)$, removes an element from the multiset, strips off the time-stamp and passes the packet p to the user (the server TCP host). The internal actions *duplicate*$_{cs}$*(p,t)* and *drop*$_{cs}$*(p,t)* duplicates and removes elements of *in-transit*$_{cs}$ respectively. The precondition on the time passage action $\nu(t)$ ensures that packets that have been in the channel for the MSL, get dropped from *in-transit*$_{cs}$.

Chapter 6

Transmission Control Protocol

In this chapter we give the formal presentation of TCP. It is specified as a general timed automaton (GTA). The automaton is the timed composition of four component automata, client and server automata, and the two channel automata described in the previous chapter. Figure 6-1 shows the composed system. In this presentation we assume the client and server have stable and unbounded counters for sequence number generation. In Chapter 8 we discuss the necessary modifications and timing assumptions needed to make TCP work correctly with non-stable and bounded counters.

Before we describe the TCP automaton, we show in Figure 6-2 a slightly simplified version of the TCP finite state machine (FSM) from the Internet Standard [28]. The simplification comes from the fact that we do not allow both sides to simultaneously try to initiate the connection as is permitted in [28]. The FSM is not meant to capture TCP in its entirety, but mainly to show the *state* changes of the client and server TCP's. Our formal presentation has more of the details of how the protocol works, but we show the standard FSM here to give some intuition for reading the steps of our automata. Since the client and server TCP's go through a lot of the same state changes, there is not a separate FSM for both sides. Instead, one can trace through the states of the client from open to close, and then separately trace out states of the server from open to close on the same FSM. The paths are not the same, but overlap in many states. They both start in `closed`, but the client goes from `closed` to `syn-sent` when it receives the signal to open (active open). This cause the client to send a SYN segment to the server. When it receives an

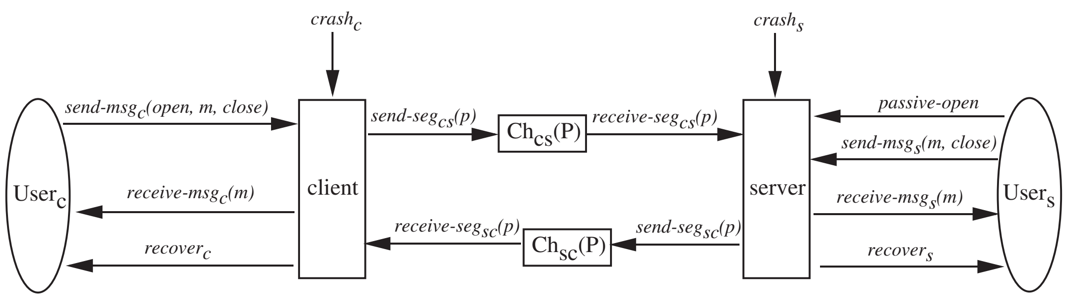

Figure 6-1: *The user interface for TCP*

acknowlegment of its SYN segment from the server, it goes to state `estb`. The server on the other hand, goes from `closed` to `listen` when it receives the signal to open (passive open). It goes from `listen` to `syn-rcvd` when it receives the SYN segment form the client. Receiving the SYN segment causes the server to send an acknowlegment and its own SYN segment. The server goes to `estb` when it receives from the client the acknowledgment of its SYN segment. In state `estb` is where data transfer occurs. After state `estb` they both can take any of possible paths shown in the figure back to state `closed`.

6.1 The formal model

We specify the client and server as GTA $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ respectively. Even though both client and server can send and receive data, they are not quite symmetric since the client side always initiates communication. Figure 6-1 shows the user interfaces of both.

6.1.1 States and start states

When the variable $mode_c$ = `closed` this means that there is no transmission control block (TCB) for the client side, which means all the client variables except sn_c which holds the sequence number and now_c which holds the current time, are undefined. Similarly, on the server side, when $mode_s$ = `closed` the TCB on the server side is also undefined. In the start state of the client automaton $\mathcal{TCP}_c$, $mode_c$ = `closed`, $sn_c = 0$, $now_c = 0$, and all the other variables are undefined; and in the start state of the server automaton $\mathcal{TCP}_s$, $mode_s$ = `closed`, $sn_s = 0$, $now_s = 0$, and all the other variables are undefined. For both $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ we use the set *Msg* to represent the set of possible messages. That is, the set

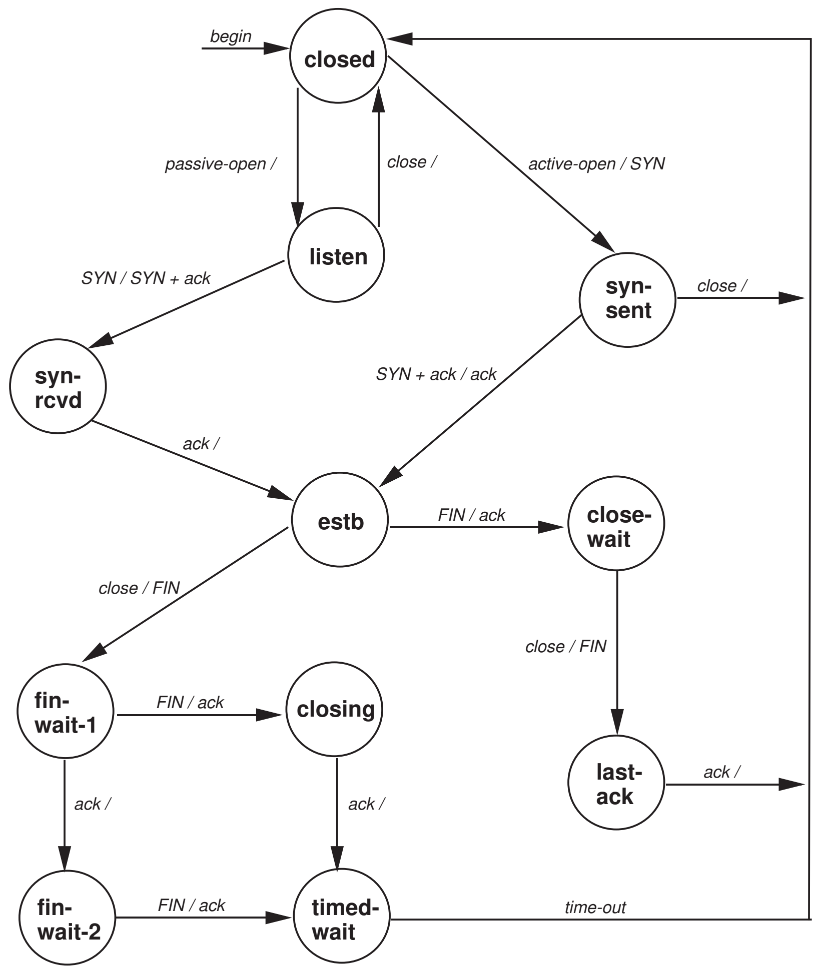

Figure 6-2: *The TCP finite state machine. Each host begins in the closed state. Labels on transitions show the input that caused the transition followed by the output if any.*

Msg is the set of all possible strings over some basic message alphabet that does not include the special symbol `null`. The symbol `null` indicates the absence of a message. The type `T` ranges over the nonnegative real numbers and represents time, and the type `N` ranges over the non-negative integers.

The first table below summarizes the type definitions. In the other tables below we describe the variables of $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$. A check in the **S** column means the variable is stable and is unaffected by crashes. As we said in the previous paragraph the non-stable variables are undefined in the start state. However, they are initialized when when the respective host opens. The initial values given in the tables for non stable variables, are values given to these variables when the transmission control blocks are initialized.

Type definitions

Type	Description
Msg	The set of all possible strings over some basic message alphabet that does not include the special symbol `null`.
T	The nonnegative real numbers — represents real time.
N	The set of non-negative integers.

Client variables

Variable	Type	S	Initially	Description
$mode_c$	{`closed`, `syn-sent`, `estb`, `fin-wait-1`, `fin-wait-2`, `close-wait`, `last-ack`, `closing`, `timed-wait`, `rec`, `reset`}		`closed`	The modes of the client. Mode `closed` indicates that the connection is closed, `syn-sent` indicates that the client has begun the synchronization process, `estb` indicates that the connection is established between the client and the server, `fin-wait-1` indicates the host has received the *close* input after the connection has been established and before receiving a FIN from the other host, `fin-wait-2` indicates that the host has received an ACK for its FIN, but before receiving a FIN, `closing` indicates it received a FIN after it sent it's FIN, but before receiving an ACK for its FIN, `close-wait` indicates that the host has received a FIN while it was in `estb` mode, `last-ack` indicates that the host has received a FIN and is waiting for an ACK to its FIN after which it closes, `timed-wait` indicates that the host has received both a FIN and an ACK(FIN) from the other host and will close in 2× MSL, `rec` indicates the client is recovering from a crash, and `reset` indicates that the client has received a valid reset and will close.
ack_c	N ∪ {`nil`}		`nil`	The acknowledgment number.
$rst\text{-}seq_c$	N∪ {`nil`}		`nil`	The number assigned to a reset segment.
$ready\text{-}to\text{-}send_c$	Bool		`true`	A flag that when true indicates that the next segment can be sent.
$send\text{-}ack_c$	Bool		`false`	A flag that enables the sending of an acknowledgment.
$send\text{-}fin_c$	Bool		`false`	A flag that enables the sending of a *FIN* segment.
$rcvd\text{-}close_c$	Bool		`false`	A flag that is set to true when the signal *close* is received.
$push\text{-}data_c$	Bool		`false`	A flag that forces the client to only perform the $receive\text{-}msg_c(m)$ action until $rcv\text{-}buf_c$ is empty after a *FIN* segment is received.
msg_c	*Msg* ∪ {`null`}		`null`	The current message to be sent.

Client variables

Variable	Type	S	Initially	Description
send-fin-ack$_c$	Bool		`false`	A flag that is set to true when the client acknowledges a FIN from mode `closing`.
send-rst$_c$	Bool		`false`	A flag that enables the sending of a reset segment.
now$_c$	`T`	$\surd$	0	The clock variable.
first(t-out$_c$*)*	$\mathtt{T} \cup \infty$		∞	The lower bound on when the client can close after starting `timed-wait` state.
time-sent$_c$	$\mathtt{T} \cup \infty$		0	Used to mark the time a segment is sent, so that the segment can be resent after RTO if it is not acknowledged.
send-buf$_c$	*Msg**		ϵ	The client buffer for messages to be sent.
rcv-buf$_c$	*Msg**		ϵ	The client buffer for messages received.
sn$_c$	`N`	$\surd$	0	Client side sequence number.

Server variables

Variable	Type	S	Initially	Description
mode$_s$	{`closed`, `listen`, `estb`, `syn-rcvd`, `fin-wait-1`, `fin-wait-2`, `close-wait`, `last-ack`, `timed-wait`, `rec`}		`closed`	The server modes. Modes `closed`, `estb`, `fin-wait-1`, `fin-wait-2`, `close-wait`, `last-ack`, `timed-wait`, and `rec` indicate the same behavior as in the client. Mode `listen` indicates the server has received a *passive-open* input and is waiting for a SYN segment from a client, mode `syn-rcvd` indicates the server has received the initial SYN segment.
send-buf$_s$	*Msg**		ϵ	The buffer for messages to be sent.
rcv-buf$_s$	*Msg**		ϵ	The buffer for messages received.
sn$_s$	`N`	$\surd$	0	Server side sequence number.
ack$_s$	`N` $\cup$ {`nil`}		`nil`	The acknowledgment number.
rst-seq$_s$	`N` $\cup$ {`nil`}		`nil`	Symmetric to *rst-seq*$_c$.
ready-to-send$_s$	Bool		`true`	Symmetric to *ready-to-send*$_c$.
send-ack$_s$	Bool		`false`	Symmetric to *send-ack*$_c$.
send-fin$_s$	Bool		`false`	Symmetric to *send-fin*$_c$.
rcvd-close$_s$	Bool		`false`	Symmetric to *rcvd-close*$_c$.
push-data$_s$	Bool		`false`	Symmetric to *push-data*$_c$.
send-fin-ack$_s$	Bool		`false`	Symmetric to *send-fin-ack*$_c$
send-rst$_s$	Bool		`false`	Symmetric to *send-rst*$_c$
now$_s$	`T`	$\surd$	0	The clock variable.
first(t-out$_s$*)*	$\mathtt{T} \cup \infty$		∞	Symmetric to *first(t-out*$_c$*)*.
time-sent$_s$	$\mathtt{T} \cup \infty$		0	Symmetric to *time-sent*$_c$.

6.1.2 Action signature

Client, $\mathcal{TCP}_c$
Input:
  $send\text{-}msg_c(open,\ m,\ close)$
    $open,\ close \in$ Bool, $m \in Msg \cup \{\texttt{null}\}$
  $receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$
  $receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$
  $receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$
  $receive\text{-}seg_{sc}(RST,\ rst\text{-}seq_s)$
  $crash_c$
Output:
  $receive\text{-}msg_c(m)\ m \in Msg$
  $send\text{-}seg_{cs}(SYN,\ sn_c)$
  $send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$
  $send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$
  $send\text{-}seg_{cs}(RST,\ rst\text{-}seq_c)$
  $recover_c$
Internal:
  $time\text{-}out_c$
  $prepare\text{-}msg_c$
  $shut\text{-}down_c$
Time-passage:
  $\nu(t),\ t \in R^+$

Server, $\mathcal{TCP}_s$
Input:
  $passive\text{-}open$
  $send\text{-}msg_s(m,\ close)\ m \in Msg$
  $receive\text{-}seg_{cs}\ (SYN,\ sn_c)$
  $receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$
  $receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$
  $receive\text{-}seg_{cs}(RST,\ rst\text{-}seq_c)$
  $crash_s$
Output:
  $receive\text{-}msg_s(m),\ m \in Msg$
  $send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$
  $send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$
  $send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$
  $send\text{-}seg_{sc}(RST,\ rst\text{-}seq_s)$
  $recover_s$
Internal:
  $time\text{-}out_s$
  $prepare\text{-}msg_s$
  $shut\text{-}down_s$
Time-passage:
  $\nu(t)\ ,\ t \in R^+$

6.1.3 Steps

From Figure 6-2 it is easy to see that the modes that require `estb` as an antecedent are `fin-wait-1`, `fin-wait-2`, `close-wait`, `closing`, `last-ack`, and `timed-wait`. Mode `estb` and these modes are said to be synchronized states, because if the hosts are in any of these states then, barring a crash, the endpoints are synchronized. In the steps of automaton $\mathcal{TCP}$ below, and for the rest of this thesis we denote the set, {`estb`, `fin-wait-1`, `fin-wait-2`, `close-wait`, `closing`, `last-ack`, `timed-wait`}, as *sync-states* — the set of synchronized states.

The steps for the timed automata for the client and server are shown in Figures 6-3, 6-4, 6-5, 6-6, and 6-7. In each of these figures the client actions are on the left and the server actions are on the right. The *receive-seg(p)* actions are shown opposite the corresponding *send-seg(p)* actions, and symmetric internal actions are opposite each other.

The open phase

The protocol starts when the client receives the action *send-msg*$_c$*(open, m, close)* (Figure 6-3) with *open* set to true (an *active open*) and the server receives a *passive-open* (Figure 6-3) input. These two actions signal that both hosts can try to establish a connection with each other. The active open and passive open are only valid if the hosts are closed. That is, the client and server can only accept inputs from the users to start a new incarnations if they are closed. When the client receives the active open it changes $mode_c$ to `syn-sent`. The client also chooses an initial sequence number (ISN) by incrementing the sn_c. This number is incremented for each segment sent, except acknowledgment only segments, through the life of the connection, and is used to order the sequence of messages. Note that the server does not choose an ISN during passive open. The *send-msg*$_c$*(open, m, close)* action might also have data to be sent. If this is the case, the data is appended to the queue *send-buf*$_c$ which is where the client keeps messages to be sent. If *close* is true this means the connection should be closed and no more data should be accepted from the user to be sent. We discuss what happens then in the discussion of the close phase.

Assuming the client does not open and close without receiving any data, the client performs the action *send-seg*$_{cs}$*(SYN,* sn_c*)* (Figure 6-3), where sn_c is the ISN, as the first step of the three-way handshake. Note that this action, along with all the other *send-seg(p)* actions have as a part of the precondition the predicate ($now_c - time\text{-}sent_c \geq \mathrm{RTO}$). This predicate controls the frequency of retransmission. That is, if an acknowledgment is not received for a segment, at least RTO must elapse before the segment is retransmitted. When this segment is received by the server, if it is in mode `listen`, it changes to mode `syn-rcvd`, chooses its ISN by incrementing sn_s, and also records the next sequence number it expects from the client, sn_c+1, in the variable ack_s. If $mode_s$ is `closed`, *send-rst*$_s$ and *rst-seq*$_s$ are set to generate a reset segment. Throughout the the protocol, whenever either host receives a segment when it is `closed`, or if it is in an unsynchronized mode (`syn-sent`, `listen`, or `syn-rcvd`) and it gets an invalid segment, a reset segment is generated. After it receives the first segment of the three-way handshake, the server performs the action *send-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)* (Figure 6-3) which is the second segment of the three-way handshake. In this segment sn_s is the server's ISN. When the client receives this segment, if a reset is not generated, it sets *send-ack*$_c$ to `true` to enable the acknowledgment of this segment. If the client is already in a synchronized state, this segment is either a duplicate created by the channel or a retransmission of a segment previously acknowledged. Since the acknowledgment might not have been received by the server, the retransmission of the acknowledgment is enabled.

If the client is in mode `syn-sent`, and $ack_s = sn_c + 1$, then it knows the server received its correct ISN, and that the segment is an acknowledgment of the SYN segment it sent. Thus, the client sets $mode_c$ to `estb` and makes assignments in preparation of sending the final segment in the three-way handshake. First ack_c is set to $sn_s + 1$ for the next expected segment, and *time-sent*$_c$ gets set to 0. Then if there is data to be sent, *send-ack*$_c$ is set to `false`, so that the acknowledgment is not sent until the data is prepared. If there is data in *send-buf*$_c$, the *prepare-msg*$_c$ (Figure 6-5) action increments the sequence number sn_c, sets *ready-to-send*$_c$ to `true` and moves the head of the send buffer to msg_c which gets sent with the next segment. The final part of the three-way handshake is the action *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* (Figure 6-4) or if the client had received a *close* input and had no more data to send, *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)* (Figure 6-6). Both segments acknowledge the SYN segment from the server. In the open phase, when the server receives this input, $mode_s$ is `syn-rcvd` and it then changes to `estb`. If there is valid data in the segment, that is $sn_c = ack_s$, it is placed on the receive buffer and ack_s is incremented. These actions are discussed in more detail in sections on the the data transfer and close phases.

The data transfer phase

Data transfer is bi-directional, but since it is also symmetric we only discuss what happens as data goes from client to the server. In this phase both the client and the server are in *mode* `estb`. The client gets data from the user from the input action *send-msg*$_c$*(open, m, close)* when m is not `null`. This data is appended to the client's send buffer. To prepare data for sending, the internal action *prepare-msg*$_c$ increments the segment number, sets *ready-to-send*$_c$ to true, and takes the first piece of data off the send buffer. If the buffer becomes empty and a *close* input had been received (*rcvd-close*$_c$ is `true`), the client begins the close phase which we discuss in the next subsection. The setting of *ready-to-send*$_c$

coupled with the fact that the client is in *mode* `estb` enables the $send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ action. This action sends the data and if there was data from the server to be acknowledged, then ack_c does that. In addition to the condition for retransmission, this actions has as part of it precondition that $ready\text{-}to\text{-}send_c \lor send\text{-}ack_c$ be true, and that the client be in a synchronized state and that $push\text{-}data_c$ be false. The condition of $push\text{-}data_c$ is explained in the section on the close phase. The condition $ready\text{-}to\text{-}send_c \lor send\text{-}ack_c$ is there because the action can be sending data just data or data and a valid acknowledgment, in which case $ready\text{-}to\text{-}send_c$ is `true`, or the segment could just have a valid acknowledgment. In this case $ready\text{-}to\text{-}send_c$ is false and $send\text{-}ack_c$ is `true`. The action sets $time\text{-}sent_c$ to the current time to start the retransmission timeout timer. Additionally, the action sets $send\text{-}ack_c$ to `false`, so that if the segment has no valid data, that is, it is just for the purposes of acknowledgment, then it does not get retransmitted. If $mode_c$ = `timed-wait` or `closing` then additional assignments are made. We discuss these assignments in the discussion of the close phase.

When the server gets the input $receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ (recall we are discussing the case where $mode_s \in sync\text{-}states$), it sets $send\text{-}ack_s$ to `true` to enable the retransmission of the previous acknowledgment the server sent. It does not matter if the segment contains valid data or not. If the data it contains is invalid, that is, the server has already sent an acknowledgment for that data, then the fact that the server receives the segment again could me that the acknowledgment for the data was not received by the client, so the server will send it again. This is the only time acknowledgments are retransmitted.

If the data is valid ($sn_c = ack_s$), it is enqueued on the server's buffer for incoming messages, $rcv\text{-}buf_s$. Additionally, ack_s is incremented for the acknowledgment of this data. If the segment contains a valid acknowledgment ($ack_c = sn_s + 1$), then $ready\text{-}to\text{-}send_s$ is set to `false` to stop the retransmission of the message acknowledged. If there is at least one message on the send buffer, $send\text{-}ack_s$ is set to `false`. The setting of $send\text{-}ack_s$ to `false` is done to delay the action $send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$ (Figure 6-5) until data is prepared to be sent. The acknowledgment piggy-backs on the data segment. If the server does not have any data to send when it received the segment, or if the segment did not contain a valid acknowledgment, only valid data, then $send\text{-}ack_s$ remains true and $send\text{-}seg_{sc}(sn_s,$

ack_s, msg_s) contains a valid acknowledgment and retransmitted data. If when the server received the segment it contains duplicated data, but a valid acknowledgment, the server sends a segment with new data if it has any to send, and an acknowledgment of the last valid data it received. Data received by the server is passed to the user by the output action *receive-msg*$_s(m)$ (Figure 6-4). If *rcv-buf*$_s$ is empty and *push-data*$_s$ is `true` then *push-data*$_s$ is set to `false` to enable other actions. The flag *push-data*$_s$ is set to true when the server receives a valid *FIN* segment. This segment means the client has stopped sending data, and that the server should send pass all the data it has to the user before doing anything else. When the client receives the input action *receive-seg*$_{sc}$(sn_s, ack_s, msg_s) (Figure 6-5), it behaves in a manner symmetric to the server's behavior when it received *receive-seg*$_{cs}$(sn_c, ack_c, msg_c).

The close phase

If the client receives the signal to close while it is still in mode `syn-sent` and its send buffer is empty, it closes immediately. Similarly, if the server receives the signal to close while it is in *mode* `listen` and its send buffer is empty, it also closes immediately. Otherwise, both sides go through a sequence of steps to ensure a "graceful" close.

Either or both sides can initiate the close sequence. The hosts go through a series of modes, that depend on the order in which they receive the *close* input from the user and FIN segments from each other. The *close* signal from the user means it has stopped sending data, but can still receive it.

When the client receives the input to close, *send-msg*$_c$(*open, m, true*), it does not immediately change modes. Instead the flag *rcvd-close*$_c$ is set to `true` to mark that the close signal has been received, while still allowing the client to proceed until it is ready to begin the close sequence. Thus, if *mode*$_c$ is `syn-sent` and the send buffer is not empty, (if the buffer is empty the client closes), the client goes through the normal open and data transfer phases as described above. If *prepare-msg*$_c$ is enabled because *rcvd-close*$_c$ is true and *send-buf*$_c$ is empty, then sn_c is incremented once in the action, and indicates that the next segment sent is just a FIN segment. However, if *rcvd-close*$_c$ is true and *send-buf*$_c$ becomes empty in the *prepare-msg*$_c$ action, then sn_c is incremented twice to indicate that the segment not only contains a FIN, but also valid data. In either case $mode_c$ is set to `fin-wait-1` if it was `estb` or `last-ack` if it was `close-wait`, and the flag $send\text{-}fin_c$ is set to `true`. The client is in mode `close-wait` if it has already received a FIN segment from the server. These assignments enable the action $send\text{-}seg_{cs}$ $(sn_c,\ ack_c,\ msg_c,\ FIN)$. The client is in mode `closing` if it received a FIN segment from the server after going to `fin-wait-1` in the $prepare\text{-}msg_c$ action, but before it performed $send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$. When the server receives this segment, it behaves almost the same as it does when it receives the $(sn_c,\ ack_c,\ msg_c)$ segment. However, the mode changes are different, and this segment is valid if sn_c is $\geq ack_s$. Also when the segment is received $push\text{-}data_s$ is set to `true` to disable all local server action until it has send all the data to the user.

If the segment also contains a valid acknowledgment, then if $mode_s$ is `syn-rcvd`, it is set to `estb`, and if it is `fin-wait-1`, it is set `fin-wait-2`. Also if there is more data to be sent from the server, the assignments are done for that. If the segment had valid data, whether it had a valid acknowledgment or not, that data is placed on $rcv\text{-}buf_s$. Also the fact that the segment is a valid FIN causes $mode_s$ to change to `close-wait` if it was previously `estb`, `closing` if it was previously `fin-wait-1`, and `timed-wait` if it was previously `fin-wait-2`. The server responds with either the action $send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$ or $send\text{-}seg_{sc}$ $(sn_s,\ ack_s,\ msg_s,\ FIN)$ (Figure 6-7). If the server sends the $send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$ segment from mode `closing` then $send\text{-}fin\text{-}ack_s$ is set to `true` to indicate that the acknowledgment of the FIN has been sent. When the client receives this segment, it if it is in mode `fin-wait-1`, it changes to mode `fin-wait-2`. If it is in mode `last-ack`, it closes. It can close from mode `last-ack` because this mode means it has sent and received a FIN segment and is only waiting for an acknowledgment of the FIN segment it sent. Since this segment is an acknowledgment of that FIN, it closes. If the client had received a FIN segment from the server before it received this segment, then it would have gone from mode `fin-wait-1` to `closing`. Now when it receives the acknowledgment, it goes to mode `timed-wait` and if it already sent the acknowledgment for that FIN, that is, if $send\text{-}fin\text{-}ack_c$ is `true`, it sets $first(t\text{-}out_c)$ to $now_c + 2\mu$. Timed-wait state ensures the graceful close property because the host that is in timed-wait state waits long enough, so that if the host that sent the final piece of data did not receive the acknowledgment for that data and retransmits that

FIN segment, the host in timed-wait state receiving the retransmitted FIN segment can retransmit the acknowledgment.

After waiting for a period of 2μ a host in mode `timed-wait` times out and closes. The actions are $timeout_c$ and $timeout_s$ (Figure 6-6) for the client and server respectively.

Other actions

The actions $send\text{-}seg_{cs}(RST,\ rst\text{-}seq_c)$ and $send\text{-}seg_{sc}(RST,\ rst\text{-}seq_s)$ (Figure 6-7) are enabled when client or server host respectively receives an inappropriate segment while in a non synchronized state. The variables $rst\text{-}seq_c$ and $rst\text{-}seq_c$ are use to validate the segments respectively. When the client or server receives a valid reset segment, it sets $mode_c$ or $mode_c$ respectively to `reset`. The setting of $mode_c$ or $mode_c$ to `reset` enables the $shut\text{-}down_c$ or $shut\text{-}down_s$ actions respectively (Figure 6-7) which causes the resepective host to close.

Crash action $crash_c$ (Figure 6-6) causes the client to change $mode_c$ `rec`. In TCP with bounded and unstable counters, a *quiet time* of qt is observed after a crashes to ensure that after recovery there are no old duplicate packets are in the network. However, since crashes do not affect the counters in formal model of TCP we are presenting, quiet time is not needed.

The corresponding recovery action $recover_c$ closes the client. The crash and recovery action are symmetric for the server.

6.2 The specification of the TCP automaton

As depicted in Figure 6-1, the TCP automaton consists of the client, the server and the two channels, so we first define $\mathcal{TCP}'$ to be the parallel composition of these automata. That is,

$$\mathcal{TCP}' \triangleq \mathcal{TCP}_c \| \mathcal{TCP}_s \| Ch_{cs}(\mathcal{P}) \| Ch_{sc}(\mathcal{P}).$$

The set $\mathcal{P}$ of possible packets of the channels is instantiated with the packets that $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ can send and receive. By the definition of parallel composition, the different $send\text{-}seg_{cs}(p)$ and $send\text{-}seg_{sc}(p)$ actions of $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ respectively and the $receive\text{-}seg_{cs}(p)$ and the $receive\text{-}seg_{sc}(p)$ actions of $Ch_{cs}(\mathcal{P})$ and $Ch_{sc}(\mathcal{P})$ respectively are output actions in $\mathcal{TCP}'$. Since these actions are not output actions in the specifications S and D, we need to hide these actions. Thus, we use the action hiding operator defined in Chapter 3. Let

$$
\begin{aligned}
\mathcal{A}_T \triangleq\ & \{\textit{receive-seg}_{sc}(SYN,\ ack_s,\ sn_s)\} \cup \\
& \{\textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s)\} \cup \\
& \{\textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)\} \cup \\
& \{\textit{receive-seg}_{sc}(RST,\ \textit{rst-seq}_s)\} \cup \\
& \{\textit{send-seg}_{cs}(SYN,\ sn_c)\} \cup \\
& \{\textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c)\} \cup \\
& \{\textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)\} \cup \\
& \{\textit{send-seg}_{cs}(RST,\ \textit{rst-seq}_c)\} \cup \\
& \{\textit{receive-seg}_{cs}\ (SYN,\ sn_c)\} \cup \\
& \{\textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c)\} \cup \\
& \{\textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)\} \cup \\
& \{\textit{receive-seg}_{cs}(RST,\ \textit{rst-seq}_c)\} \cup \\
& \{\textit{send-seg}_{sc}(SYN,\ sn_s,\ ack_s)\} \cup \\
& \{\textit{send-seg}_{sc}(sn_s,\ ack_s,\ msg_s)\} \cup \\
& \{\textit{send-seg}_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)\} \cup \\
& \{\textit{send-seg}_{sc}(RST,\ \textit{rst-seq}_s)\}
\end{aligned}
$$

The complete general time automaton model for TCP, $\mathcal{TCP}$, is defined as:

$$\mathcal{TCP} \triangleq \mathcal{TCP}' \backslash \mathcal{A}_T.$$

This definition gives a timed automaton with the same set of input and output actions as S and D which is necessary for doing a simulation from $\mathcal{TCP}$ to any of the specifications.

6.3 Derived variables for $\mathcal{TCP}$

We define four derived variables for $\mathcal{TCP}$. These variables are needed for the verification of TCP which we present in the next chapter.

The first two derived variables are $\textit{cur-msg}_c$ and $\textit{cur-msg}_s$. These are the "current message" being sent by the client and server respectively. They are defined as follows.

$$s.cur\text{-}msg_c \triangleq \begin{cases} (s.msg_c, \mathtt{ok}) & \text{if } s.mode_c \notin \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}, \mathtt{syn\text{-}sent}\} \wedge \\ & \quad ((s.sn_c = s.ack_s) \wedge \neg(s.rcvd\text{-}close_c \wedge s.send\text{-}buf_c \ = \epsilon)) \\ & \quad \vee (s.sn_c = s.ack_s + 1) \\ \epsilon & \text{otherwise} \end{cases}$$

$$s.cur\text{-}msg_s \triangleq \begin{cases} (s.msg_s, \mathtt{ok}) & \text{if } s.mode_s \notin \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}, \mathtt{listen}, \mathtt{syn\text{-}rcvd}\} \wedge \\ & \quad ((s.sn_s = s.ack_c) \wedge \neg(s.rcvd\text{-}close_s \wedge s.send\text{-}buf_s \ = \epsilon)) \\ & \quad \vee (s.sn_s = s.ack_c + 1) \\ \epsilon & \text{otherwise} \end{cases}$$

The current message is the message that is about to be sent or is being sent, but has not yet been received. For the client side the condition $s.sn_c = s.ack_s \wedge \neg(s.rcvd\text{-}close_s \wedge s.send\text{-}buf_s \ = \epsilon)$ holds when a message on a non FIN segment has not yet being received. If the current message is on a FIN segment, then the condition $s.sn_c = s.ack_s + 1$ holds until the message is received. When the message is received, $cur\text{-}msg_c$ message becomes the empty string, because when a message is received the acknowledgment variable is assigned the value of one plus the sequence number of the received segment. The current message derived variables are used to hold one copy of a message that might be in both a send buffer and on a channel. When the message is received, we know precisely where one copy of the message is, so we do not need the value to be held in a current message variable. In the next section when we define the refinment mapping, we will see why this is useful. The message is paired with the value ok, to match variables on the queues in D, which again is needed for the refinement mapping.

The next two variables we define are $p\text{-}pair_c$ and $p\text{-}pair_s$ for the client and server side respectively. These variables are "possible pairs." The term "pair" is used because the variables are sets of pairs. Each pair is of the form (i, m) where i is a sequence number, and m is message. That is, $i \in \mathbb{N}$ and $m \in Msg$. The term "possible" is used because the sequence number and message that form a pair comes from segments that are on the channel, where there is a possibility that the message m may or may not get delivered. These segments exists when the sender of the segment crashes before the segment is received. That is, they are segments that contain the message from the "current message" derived variables, if the sender crashes or resets. There is possibility that the message may not be delivered because the sender can no longer retransmit the segments that contain the message. Therefore, if all copies of segments with this message get dropped the channel, then the message will not be delivered. Also if the receiving host crashes or closes before the message is received, then the message will also not get delivered. If all copies do not get dropped, and the receiving host does not crash or close, then the message will get delivered. Recall that in Chapter 4 the reason we gave for defining the Delay Decision Specification, was that in the low level protocols, whether a message gets lost because of crash may not be determined until after recovery. We define possible pairs such that the messages in the pairs are messages that may be lost after a crash and recovery. The formal definitions are given below.

For any segment p on $\mathit{in\text{-}transit}_{cs}$ or $\mathit{in\text{-}transit}_{sc}$, define $sn(p)$ to be the sequence number of the segment, $ack(p)$ to be the acknowledgment number of the segment, and $msg(p)$ to be the message of the segment. For example, for a segment $p = (sn_c, ack_c, msg_c)$ where $sn_c = i$ and $ack_c = j$, and $msg_c = m$, $sn(p) = i$, $ack(p) = j$, and $msg(p) = m$. If the p is of the form $(SYN,\ sn_c)$ or $(SYN,\ sn_s,\ ack_s)$ then $msg(p) = \texttt{null}$ and if p is of the form $(SYN,\ sn_c)$ then $ack(p) = \texttt{nil}$. Let s be any state in $\mathcal{TCP}$, then

$$s.p\text{-}pair_c \triangleq \begin{cases} \{(i,m) \mid \exists\, p \in s.\mathit{in\text{-}transit}_{cs} \text{ s.t. } sn(p) = i\ \wedge msg(p) = m \wedge m \neq \texttt{null}\} \\ \quad \text{if } s.mode_c \in \{\texttt{rec}, \texttt{reset}, \texttt{closed}, \texttt{syn-sent}\} \wedge \\ \quad (i = s.ack_s\ \wedge\ p \text{ is not a FIN segment}) \vee \\ \quad (i = s.ack_s + 1\ \wedge\ p \text{ is a FIN segment}) \\ \emptyset \quad \text{otherwise,} \end{cases}$$

$$s.p\text{-}pair_s \triangleq \begin{cases} \{(i,m) \mid \exists\, p \in s.\mathit{in\text{-}transit}_{sc} \text{ s.t. } sn(p) = i\ \wedge msg(p) = m \wedge m \neq \texttt{null}\} \\ \quad \text{if } s.mode_s \in \{\texttt{rec}, \texttt{reset}, \texttt{closed}, \texttt{listen}, \texttt{syn-rcvd}\} \wedge \\ \quad (i = s.ack_c\ \wedge\ p \text{ is not a FIN segment}) \vee \\ \quad (i = s.ack_c + 1\ \wedge\ p \text{ is a FIN segment}) \\ \emptyset \quad \text{otherwise.} \end{cases}$$

The variables are sets of pairs as opposed to just being segments or sets of segment because there can be segments on the channels with the same sequence numbers and messages, but different acknowledgment numbers. Invariant 7.37, which we define in the next chapter, tells us that segments with the same sequence number must have the same message.

Therefore, the possible pair set has one element for each message that might get delivered. If we use segments or sets of segments in the definitions this would not be the case. Because in our model of TCP each message must receive an acknowledgment before another is sent, we can show that $p\text{-}pair_c$ and $p\text{-}pair_s$ always have at most one element. The this claim follows from Invariants 7.37, 7.62 and 7.64 also defined in the next chapter.

There are two types of segments with messages — segments with or without the *FIN* bit. That is why we have the two cases (aside from the empty case) in the definition of possible segments. In order for the message on a non FIN segment to be accepted, its sequence number must be equal to the acknowledgment number of the receiving host, and in order for the message on a FIN segment to be accepted, its sequence number must be equal to the acknowledgment number plus one of the receiving host. Also a message on a segment might possibly be delivered after the sender crashes and recovers, but only if the sender has not gotten back to a synchronized state after the crash. This is the case because after a host crashes, TCP forces the other side to reset before they can both be synchronized again. Therefore, if the sender got back to a synchronized state, it means a new incarnation has started, so the message cannot be delivered because it is from the previous incarnation.

In the next chapter were we present the verification of $\mathcal{TCP}$, we use the operator *data* to extract the the message from element of the possible pair set, so for example if $s.p\text{-}pair_c = \{(s.msg_c, s.sn_c)\}$, then $data(s.p\text{-}pair_c) = s.msg_c$. If $s.p\text{-}pair = \{\}$ then $data(s.p\text{-}pair_c) = \epsilon$.

The formal modeling of TCP is now complete, and in the next chapter we prove that TCP implements a patient version of our specification S.

$send\text{-}msg_c(open,\ m,\ close)$

Eff: if $mode_c = \texttt{closed} \wedge open$ then $\{$
  initialize TCB_c
  $time\text{-}sent_c := 0$
  $mode_c := \texttt{syn-sent}$
  $sn_c := sn_c + 1$
$\}$
if $\neg rcvd\text{-}close_c \wedge\ m\ \neq \texttt{null} \wedge$
$mode_c \in \{\texttt{syn-sent}, \texttt{estb}, \texttt{close-wait}\}$
  then $send\text{-}buf_c := send\text{-}buf_c \cdot m$
if $close$ then $\{$
  $rcvd\text{-}close_c :=$ true
  if $mode_c = \texttt{syn-sent} \wedge send\text{-}buf_c = \epsilon$
    then $mode_c := \texttt{closed}$
$\}$

$send\text{-}seg_{cs}(SYN,\ sn_c)$

Pre: $(now_c$ - $time\text{-}sent_c \geq \mathrm{RTO}) \wedge$
  $mode_c = \texttt{syn-sent} \wedge \neg\ send\text{-}rst_c$
Eff: $time\text{-}sent_c := now_c$

$receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$

Eff: if $(mode_c = \texttt{closed}) \vee$
  $(mode_c = \texttt{syn-sent}\ \wedge ack_s \neq sn_c + 1)$
  then $\{$
    $send\text{-}rst_c := \texttt{true}$
    $rst\text{-}seq_c := ack_s$
  $\}$
  else $\{$
    $send\text{-}ack_c := \texttt{true}$
    if $mode_c = \texttt{syn-sent}$ then $\{$
      $mode_c := \texttt{estb}$
      $ack_c := sn_s + 1$
      $time\text{-}sent_c := 0$
      $ready\text{-}to\text{-}send_c := \texttt{false}$
      if $send\text{-}buf_c \neq \epsilon$ then
        $send\text{-}ack_c := \texttt{false}$
    $\}$
  $\}$

$passive\text{-}open$

Eff: if $mode_s = \texttt{closed}$ then $\{$
  initialize TCB_s
  $mode_s := \texttt{listen}$
$\}$

$send\text{-}msg_s(m,\ close)$

Eff: if $\neg rcvd\text{-}close_s \wedge\ m\ \neq \texttt{null} \wedge$
  $mode_s \in \{\texttt{syn-rcvd}, \texttt{estb}, \texttt{close-wait}\}$
  then $send\text{-}buf_s := send\text{-}buf_s \cdot m$
if $close$ then $\{$
  $rcvd\text{-}close_s := \texttt{true}$
  if $mode_s = \texttt{listen} \wedge send\text{-}buf_s = \epsilon$
    then $mode_s := \texttt{closed}$
$\}$

$receive\text{-}seg_{cs}\ (SYN,\ sn_c)$

Eff: if $mode_s = \texttt{listen}$ then $\{$
  $mode_s := \texttt{syn-rcvd}$
  $sn_s := sn_s + 1$
  $ack_s := sn_c + 1$
  $time\text{-}sent_s := 0$
$\}$
if $mode_s = \texttt{closed}$ then
  $send\text{-}rst_s := \texttt{true}$
  $rst\text{-}seq_s := 0$
  $ack_s := sn_c + 1$

$send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$

Pre: $(now_s$ - $time\text{-}sent_s \geq \mathrm{RTO}) \wedge$
  $mode_s = \texttt{syn-rcvd} \wedge \neg\ send\text{-}rst_s$
Eff: $time\text{-}sent_s := now_s$

Figure 6-3: *Steps from the open phase of* $\mathcal{TCP}_c$ *and* $\mathcal{TCP}_s$*. The client steps are on the left and the corresponding server steps are on the right.*

send-seg$_{cs}$(sn_c, ack_c, msg_c)
Pre: $(now_c - time\text{-}sent_c \geq RTO) \wedge$
$(ready\text{-}to\text{-}send_c \vee send\text{-}ack_c) \wedge$
$mode_c \in sync\text{-}states \wedge \neg push\text{-}data_c$
Eff: $time\text{-}sent_c := now_c$
$send\text{-}ack_c :=$ `false`
if $mode_c =$ `closing` then
$send\text{-}fin\text{-}ack_c :=$ `true`
if $mode_c =$ `timed-wait` then
$first(t\text{-}out_c) := now_c + 2\mu$

receive-msg$_c(m)$
Pre: $mode_c \notin \{$`rec`, `reset`$\} \wedge rcv\text{-}buf_c \neq \epsilon$
$\wedge head(rcv\text{-}buf_c) = m$
Eff: $rcv\text{-}buf_c := tail(rcv\text{-}buf_c)$
if $push\text{-}data_c \wedge rcv\text{-}buf_c = \epsilon$ then
$push\text{-}data_c :=$ `false`

$\nu(t)$ (time-passage)
Pre: $t \in R^+$
Eff: $now_c := now_c + t$

receive-seg$_{cs}$(sn_c, ack_c, msg_c)
Eff: if $(mode_s \in \{$`closed`, `listen`$\}) \vee$
$(mode_s =$ `syn-rcvd` $\wedge\, ack_c \neq sn_s + 1)$
then {
$send\text{-}rst_s :=$ `true`
$rst\text{-}seq_s := ack_c$
}
else if $mode_s \notin \{$`rec`, `reset`$\}$ then {
if $msg_c \neq$ `null` then
$send\text{-}ack_s :=$ `true`
if $sn_c = ack_s$ then {
$ack_s := sn_c + 1$
$time\text{-}sent_s := 0$
$rcv\text{-}buf_s := rcv\text{-}buf_s \cdot msg_c$
}
if $ack_c = sn_s + 1$ then {
$msg_s :=$ `null`
$ready\text{-}to\text{-}send_s :=$ `false`
$send\text{-}fin_s :=$ `false`
if $mode_s =$ `syn-rcvd` then
$mode_s :=$ `estb`
if $send\text{-}buf_s \neq \epsilon$ then
$send\text{-}ack_s :=$ `false`
if $mode_s =$ `fin-wait-1` then
$mode_s :=$ `fin-wait-2`
if $mode_s =$ `last-ack` then
$mode_s :=$ `closed`
if $mode_s =$ `closing` then {
$mode_s :=$ `timed-wait`
if $send\text{-}fin\text{-}ack_s$ then
$first(t\text{-}out_s) := now_s + 2\mu$
}
}
}

receive-msg$_s(m)$
Pre: $mode_s \notin \{$`rec`, `reset`$\} \wedge\ rcv\text{-}buf_s \neq \epsilon$
$\wedge head(rcv\text{-}buf_s) = m$
Eff: $rcv\text{-}buf_s := tail(rcv\text{-}buf_s)$
if $push\text{-}data_s \wedge rcv\text{-}buf_s = \epsilon$ then
$push\text{-}data_s :=$ `false`

$\nu(t)$ (time-passage)
Pre: $t \in R^+$
Eff: $now_s := now_s + t$

Figure 6-4: *The basic message sending step of the client and the corresponding step to receive this segment at the server. Also the steps for $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ that pass messages to the users, and the time-passage steps.*

$receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$

```
Eff: if mode_c ∈ {closed, syn-sent} then {
          send-rst_c := true
          rst-seq_c := ack_s
     }
     else if mode_c ∉ {rec, reset} then {
       if msg_s ≠ null then
         send-ack_c := true
       if sn_s = ack_c then {
         ack_c := sn_s + 1
         time-sent_c := 0
         rcv-buf_c := rcv-buf_c·msg_s
       }
       if ack_s = sn_c + 1 then {
         msg_c := null
         ready-to-send_c := false
         send-fin_c := false
         if send-buf_c ≠ ε then
           send-ack_c := false
         if mode_c = fin-wait-1 then
           mode_c := fin-wait-2
         if mode_c = last-ack then
           mode_c := closed
         if mode_c = closing then {
           mode_c := timed-wait
           if send-fin-ack_c then
             first(t-out_c) := now_c + 2μ
         }
       }
```

$prepare\text{-}msg_c$

```
Pre: ¬push-data_c ∧ ¬ready-to-send_c ∧
     mode_c ∈ {estb, close-wait} ∧
     (send-buf_c ≠ ε ∨ rcvd-close_c)
Eff: ready-to-send_c := true
     if send-buf_c ≠ ε then {
       sn_c := sn_c + 1
       msg_c := head(send-buf_c)
       send-buf_c := tail(send-buf_c)
     }
     if rcvd-close_c ∧ send-buf_c = ε then {
       sn_c := sn_c + 1
       ready-to-send_c := false
       send-fin_c := true
       if mode_c = estb then
         mode_c := fin-wait-1
       else if mode_c = close-wait then
         mode_c := last-ack
```

$send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$

```
Pre: (now_s − time-sent_s ≥ RTO)∧
     (ready-to-send_s ∨ send-ack_s)∧
     mode_s ∈ sync-states ∧ ¬push-data_s
Eff: time-sent_s := now_s
     send-ack_s := false
     if mode_s = closing then
       send-fin-ack_s := true
     if mode_s = timed-wait then
       first(t-out_s) := now_s + 2μ
```

$prepare\text{-}msg_s$

```
Pre: ¬push-data_s ∧ ¬ready-to-send_s∧
     mode_s ∈ {estb, close-wait} ∧
     (send-buf_s ≠ ε ∨ rcvd-close_s)
Eff: ready-to-send_s := true
     if send-buf_s ≠ ε then {
       sn_s := sn_s + 1
       msg_s := head(send-buf_s)
       send-buf_s := tail(send-buf_s)
     }
     if rcvd-close_s ∧ send-buf_s = ε then {
       sn_s := sn_s + 1
       ready-to-send_s := false
       send-fin_s := true
       if mode_s = estb then
         mode_s := fin-wait-1
       else mode_s = close-wait then
         mode_s := last-ack
```

Figure 6-5: *The basic message sending step of the server and the corresponding step to receive this segment at the client. Also the steps that prepare messages to be sent.*

$send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$
Pre: $(now_c - time\text{-}sent_c \geq RTO) \land mode_c \in \{\texttt{fin-wait-1}, \texttt{last-ack}, \texttt{closing}\} \land send\text{-}fin_c \land \neg pdata_c$
Eff: $time\text{-}sent_c := now_c$

$time\text{-}out_c$
Pre: $mode_c = \texttt{timed-wait} \land now_c \geq first(t\text{-}out_c)$
Eff: $mode_c := \texttt{closed}$

$crash_c$
Eff: if $mode_c \neq \texttt{closed}$ then
    $mode_c := \texttt{rec}$

$recover_c$
Pre: $mode_c = \texttt{rec}$
Eff: $mode_c := \texttt{closed}$

$receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$
Eff: if $(mode_s \in \{\texttt{closed}, \texttt{listen}\}) \lor (mode_s = \texttt{syn-rcvd} \land ack_c \neq sn_s + 1)$
  then {
    $send\text{-}rst_s := \texttt{true}$
    $rst\text{-}seq_s := ack_c$
  }
  else if $mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ then {
    $send\text{-}ack_s := \texttt{true}$
    if $sn_c = ack_s \lor sn_c = ack_s + 1$ then {
      $push\text{-}data_s := \texttt{true}$
      $time\text{-}sent_s := 0$
      if $ack_c = sn_s + 1$ then {
        $msg_s := \texttt{null}$
        $ready\text{-}to\text{-}send_s := \texttt{false}$
        $send\text{-}fin_s := \texttt{false}$
        if $mode_s = \texttt{syn-rcvd}$ then
          $mode_s := \texttt{estb}$
        if $send\text{-}buf_s \neq \epsilon$ then
          $send\text{-}ack_s := \texttt{false}$
        if $mode_s = \texttt{fin-wait-1}$ then
          $mode_s := \texttt{fin-wait-2}$
      }
      if $sn_c = ack_s + 1$ then
      $ack_s := sn_c + 1$
        $rcv\text{-}buf_s := rcv\text{-}buf_s \cdot msg_s$
      if $mode_s = \texttt{estb}$ then
        $mode_s := \texttt{close-wait}$
      else if $mode_s = \texttt{fin-wait-1}$ then
        $mode_s := \texttt{closing}$
        else if $mode_s = \texttt{fin-wait-2}$ then
          $mode_s := \texttt{timed-wait}$
    }
  }

$time\text{-}out_s$
Pre: $mode_s = \texttt{timed-wait} \land now_s \geq first(t\text{-}out_s)$
Eff: $mode_s := \texttt{closed}$

$crash_s$
Eff: if $mode_s \neq \texttt{closed}$ then
    $mode_s := \texttt{rec}$

$recover_s$
Pre: $mode_s = \texttt{rec}$
Eff: $mode_s := \texttt{closed}$

Figure 6-6: *The steps for the client to send a FIN segment and the receiving of that segment at the server. Also the time-out steps and the crash and recovery steps.*

receive-seg$_{sc}$(sn_s, ack_s, msg_s, FIN)
Eff: if $mode_c \in$ {`closed`, `syn-sent`} then {
    $send\text{-}rst_c$:= `true`
    $rst\text{-}seq_c := ack_s$
  }
  else if $mode_c \notin$ {`rec`, `reset`} then {
    $send\text{-}ack_c$:= `true`
    if $sn_s = ack_c \vee sn_s = ack_c + 1$ then {
      $push\text{-}data_c$:= `true`
      if $mode_c$ = `estb` then
        $mode_c$:= `close-wait`
      else if $mode_c$ = `fin-wait-1` then
        $mode_c$:= `closing`
        else if $mode_c$ = `fin-wait-2` then
          $mode_c$:= `timed-wait`
      if $sn_s = ack_c + 1$ then
        $rcv\text{-}buf_c := rcv\text{-}buf_c \cdot msg_s$
      $ack_c := sn_s + 1$
      $time\text{-}sent_c := 0$
      if $ack_s = sn_c + 1$ then {
        if $mode_c$ = `closing` then
          $mode_c$:= `timed-wait`
        msg_c := `null`
        $ready\text{-}to\text{-}send_c$:= `false`
        $send\text{-}fin_c$:= `false`
        if $send\text{-}buf_c \neq \epsilon$ then
          $send\text{-}ack_c$:= `false`
      }
    }
  }

send-seg$_{sc}$(sn_s, ack_s, msg_s, FIN)
Pre: $(now_s - time\text{-}sent_s \geq RTO) \wedge mode_s \in$
  {`fin-wait-1`, `last-ack`, `closing`}
  $\wedge\ send\text{-}fin_s \wedge \neg push\text{-}data_s$
Eff: $time\text{-}sent_s := now_s$

send-seg$_{cs}$(RST, ack_c, $rst\text{-}seq_c$)
Pre: $mode_c \in$ {`closed`, `syn-sent`}
  $\wedge send\text{-}rst_c$ = `true`
Eff: $send\text{-}rst_c$:= `false`

receive-seg$_{cs}$(RST, ack_c, $rst\text{-}seq_c$)
Eff: if $mode_s \neq$ `rec` $\wedge\ rst\text{-}seq_c = ack_s$ then
    $mode_s$:= `reset`

receive-seg$_{sc}$(RST, ack_s, $rst\text{-}seq_s$)
Eff: if $mode_c \neq$ `rec` $\wedge\ rst\text{-}seq_s = ack_c \vee$
  $(rst\text{-}seq_s = 0 \wedge ack_s = sn_c + 1)$
    then $mode_c$:= `reset`

send-seg$_{sc}$(RST, ack_s, $rst\text{-}seq_s$)
Pre: $mode_s \in$ {`closed`, `listen`, `syn-rcvd`} $\wedge$
  $send\text{-}rst_s$ = `true`
Eff: $send\text{-}rst_s$:= `false`

shut-down$_c$
Pre: $mode_c$ = `reset`
Eff: $mode_c$:= `closed`

shut-down$_s$
Pre: $mode_s$ = `reset`
Eff: $mode_s$:= `closed`

Figure 6-7: *The steps for the server to send a FIN segment and the receiving of that segment at the client. Also the steps that send resets, the receiving of these resets, and the closing because of the reset.*

Chapter 7

Verification of TCP

In this chapter we prove the correctness of TCP with respect to a *patient* version of Specification S. We need to show correctness with respect to a *patient* version of S because $\mathcal{TCP}$ is a GTA and S is an untimed automaton. Instead of doing a complex backward simulation directly from $\mathcal{TCP}$ to *patient(S)*, we take the second[1] intermediate step of showing a refinement mapping from $\mathcal{TCP}$ to a patient version of D. However, we cannot find a refinement mapping from $\mathcal{TCP}$ to *patient(D)* without first adding history variables to $\mathcal{TCP}$. We call the resulting automaton $\mathcal{TCP}^h$, and we denote *patient(D)* as D^p.

7.1 $\mathcal{TCP}$ with history variables

We add several history variables to $\mathcal{TCP}$. The first two history variables we add are isn_c and isn_s. These variables correspond to id_c and id_s respectively in D, and they record the initial sequence numbers chosen by the client and the server respectively for an incarnation of the connection. These variables are not stable, but instead of being deleted with the rest of the TCB when a host closes, they take the special value `nil`. We also add the history variable isn_c^s which records the value of isn_c when the server receives a SYN segment from the client. It's symmetric counterpart is isn_s^c. Variables $used\text{-}id_c$, $used\text{-}id_s$, and $assoc$ are stable and are meant to correspond to the variables of the same names in D. We also add another stable set we call *estb-pairs* which is the set of initial sequence numbers of the client

[1]The first intermediate step being the backward simulation from D to S presented in Chapter 4.

Variable	Type	S	Initially	Description
isn_c	$\mathtt{N} \cup \mathtt{nil}$		`nil`	The initial sequence number chosen when the client opens.
isn_s	$\mathtt{N} \cup \mathtt{nil}$		`nil`	The initial sequence number chosen when the server receives a SYN segment from the client.
isn_c^s	$\mathtt{N} \cup \mathtt{nil}$		`nil`	Records the initial sequence number of the client as a server side variable.
isn_s^c	$\mathtt{N} \cup \mathtt{nil}$		`nil`	Symmetric to isn_c^s.
$used\text{-}id_c$	$2^{\mathtt{N}}$	$\surd$	$\emptyset$	The set of initial sequence numbers used by the client.
$used\text{-}id_s$	$2^{\mathtt{N}}$	$\surd$	$\emptyset$	The set of initial sequence numbers used by the server.
$assoc$	$2^{(\mathtt{N}\times\mathtt{N})}$	$\surd$	$\emptyset$	A set of pairs of *isn*'s for each incarnation of the connection.
$estb\text{-}pairs$	$2^{(\mathtt{N}\times\mathtt{N})}$	$\surd$	$\emptyset$	The set of initial sequence numbers the client has every time it reaches mode `estb`, paired with the initial sequence number received from the server.
$choose\text{-}isn_c$	Bool		`false`	A flag that is set to `true` when the client first chooses an ISN for an incarnation and set to `false` when the client sends a segment with this ISN.
$choose\text{-}isn_s$	Bool		`false`	Symmetric to $choose\text{-}isn_c$.

paired with the initial sequence number the client receives from the server after the second step of the three-way handshake. We add this set because there are executions where the client gets to mode `estb` and sends the third segment of the three-way handshake protocol, but closes before the segment is received by the server. This segment may cause the initial sequence number that the client had when it sent the segment to form an association pair with the initial sequence number of the server. Thus, *estb-pairs* records pairs that indicates the second leg of the three-way handshake as been successfully completed. We also add the history variables $choose\text{-}isn_c$ and $choose\text{-}isn_s$. These history variables are flags that become true in the step that causes the client and server respectively to choose initial sequence numbers. They become false in any subsequent steps. The table below provides more details on the history variables. Recall that the type `N` represents the set of non-negative integers.

As discussed in Chapter 3, history variables are allowed at each step to be assigned a value based on all variables in the system, but must not affect the enabledness of actions or the changes made to other (ordinary) variables. In Figure 7-1 we show where assignments to the history variables should be placed in the effect clauses of $\mathcal{TCP}$ to get $\mathcal{TCP}^h$. We omit the assignments to the original variables (by writing ... instead) but outline the if-then-else statements. The first addition is to the *send-msg*$_c$*(open, m, close)* step. The variable isn_c is assigned the value of sn_c after sn_c is incremented. The flag *choose-isn*$_c$ is also set to `true` in this step. This flag is used to indicate that the client has just chosen an initial sequence number. We use the fact that this flag is true immediately after this step and then set to false on subsequent steps, to prove certain properties about the client's initial sequence number when it is first chosen.

When the client performs the *send-seg*$_{cs}$*(SYN,* sn_c*)* action, *choose-isn*$_c$ is set to false. When the server receives a SYN segment from the client via the *receive-seg*$_{cs}$*(SYN,* sn_c*)* action, it sets *choose-isn*$_s$ to `true`, and assigns the isn_s history variable the incremented value of sn_s. The history variable isn_c^s is also assigned to $[sn_c]$ in this step. This history variable is used to record the value of what the server believes is the initial sequence number of the client. This value on the SYN segment will be paired with the new value of isn_s to form an association pair, if the received segment is indeed a valid attempt my the client to start a new incarnation and not an old duplicate. If the pair is actually added to *assoc* when the server performs either the *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* or *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)* action, the current value of isn_c might be `nil` if the client is currently closed, or it might be different from the value the server received in the *(SYN,* sn_c*)* segment, if the client closed and reopened. Therefore, in the *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)* action, when the pair is added to *assoc*, it is the pair (isn_c^s, isn_s).

After the server receives the *(SYN,* sn_c*)* segment, it responds with the *send-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)* action. In this step it sets both *choose-isn*$_c$ and *choose-isn*$_s$ to `false`. These settings, are again to facilitate the proof of invariants about the values of initial sequence numbers relative to sequence and acknowledgment numbers of other segments on the channels.

When the client receives the *(SYN,* sn_s*,* ack_s*)* segment from the server, it assigns isn_s^c to $[sn_s]$ which it believes is the initial sequence number of the server. This assignment is made if $[ack_s] = sn_c + 1$, which means the server received the correct initial sequence number of the client. It also means that the initial sequence number of the client and $[sn_s]$ will form an association pair if the third step of the three-way handshake is successful. A record of this pair is added to history variable *estb-pairs*. If the client crashes or receives a reset after it sends the third segment of the three-way handshake protocol, but before it is received by the server, neither the client nor server knows that the second leg of the three-way handshake is successful. The history variable *estb-pairs* keeps a record of this fact.

When the client performs the $receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$ actions, $choose\text{-}isn_s$ is set to `false`. Again this is to facilitate the proof of properties about initial sequence numbers.

send-msg$_c$*(open, m, close)*
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
    if $mode_c$ = **closed** $\wedge$ *open* then {
        ...
        $choose\text{-}isn_c$:= **true**
        $sn_c := sn_c + 1$
        $isn_c := sn_c$
        $used\text{-}id_c := used\text{-}id_c \cup \{isn_c\}$
        }
    ...

send-seg$_{cs}$ *(SYN,* sn_c*)*
Pre: (* Precondition clause from $\mathcal{TCP}_c$ *)
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
    $choose\text{-}isn_c$:= **false**

receive-seg$_{sc}$*(SYN,* sn_s, ack_s*)*
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
    if $mode_c$ = **syn-sent** $\wedge$ $ack_s = sn_c$+1
    then {
        $isn_s^c := sn_s$
        $estb\text{-}pairs := estb\text{-}pairs \cup \{(isn_c, isn_s^c)\}$
        ...

send-seg$_{cs}$*(*sn_c, ack_c, msg_c*)*
Pre: (* Precondition clause from $\mathcal{TCP}_c$ *)
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
    $choose\text{-}isn_s$:= **false**

send-seg$_{cs}$*(*sn_c, ack_c, msg_c*, FIN)*
Pre: (* Precondition clause from $\mathcal{TCP}_c$ *)
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
    $choose\text{-}isn_s$:= **false**

receive-seg$_{cs}$ *(SYN,* sn_c*)*
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
    if $mode_s$ = **listen** then {
        ...
        $choose\text{-}isn_s$:= **true**
        $sn_s := sn_s + 1$
        $isn_s := sn_s$
        $isn_c^s := sn_c$
        $used\text{-}id_s := used\text{-}id_s \cup \{isn_s\}$
    }
    ...

send-seg$_{sc}$*(SYN,* sn_s, ack_s*)*
Pre: (* Precondition clause from $\mathcal{TCP}_s$ *)
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
    $choose\text{-}isn_s$:= **false**
    $choose\text{-}isn_c$:= **false**
    ...

receive-seg$_{cs}$*(*sn_c, ack_c, msg_c*)*
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
    ...
    else if $mode_s \neq$ **rec** then {
        ...
        if $sn_c = ack_s$ then
            ...
            if $mode_s$ = **syn-rcvd** then {
                $assoc := assoc \cup \{(isn_c^s, isn_s)\}$
                ...

receive-seg$_{cs}$*(*sn_c, ack_c, msg_c*, FIN)*
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
    ...
    else if $mode_s \neq$ **rec** then {
        ...
        if $sn_c = ack_s \vee sn_c = ack_s + 1$ then
            ...
            if $ack_c = sn_s + 1$ then {
                ...
                if $mode_s$ = **syn-rcvd** then {
                    $assoc := assoc \cup \{(isn_c^s, isn_s)\}$
                    ...

Figure 7-1: *Steps where* $\mathcal{TCP}^h$ *differs from* $\mathcal{TCP}$*. The client side is on the left and the server side on the right.*

7.2 Invariants

During the process of performing a simulation proof it becomes clear that certain invariants are needed. This happens when the simulation relation does not hold for some situation, but it turns out the situation happens in an unreachable state. Thus, an invariant that avoids these "bad" states is found. In this section we present the invariants we need for the refinement mapping from $\mathcal{TCP}^h$ to D^p. The proofs for these invariants are given in Appendix B. We present the invariants before we present the simulation because for correctness they are needed before the simulation. However, we discovered most of the invariants we need while carrying out the simulation.[2] Some of the invariants presented below have several parts. The invariant is the conjunction of the different parts. The properties stated below are true of all reachable states of $\mathcal{TCP}^h$.

The first set of invariants, Invariants 7.1 through 7.12, state basic properties about the relationships between sequence numbers, sequence numbers on segments, acknowledgment numbers, acknowledgment numbers on segments, and initial sequence numbers. These invariants are mainly used in the proofs of other more complicated invariants.

Invariant 7.1

1. For all segments $p \in \textit{in-transit}_{cs}$, $sn_c \geq sn(p)$.

2. For all segments $p \in \textit{in-transit}_{sc}$, $sn_s \geq sn(p)$. ■

Invariant 7.2

1. If $ack_s \in \mathtt{N}$ then $ack_s \leq sn_c + 1$.

2. If $ack_c \in \mathtt{N}$ then $ack_c \leq sn_s + 1$. ■

Invariant 7.3

1. For all segments $p \in \textit{in-transit}_{sc}$, $ack(p) \leq sn_c + 1$.

2. For all segments $p \in \textit{in-transit}_{cs}$, $ack(p) \leq sn_s + 1$.

Invariant 7.4

1. If $mode_c = \texttt{syn-sent}$ then for all non-SYN segments $p \in \textit{in-transit}_{cs}$, $sn(p) < isn_c$.

[2] In terms of understanding the correctness proof in this chapter, it might be better to start reading Section 7.3 on the simulation first, and only read the invariants as they are referred to in that section.

2. If $mode_s =$ `syn-rcvd` then for all non-SYN segments $p \in in\text{-}transit_{sc}$, $sn(p) < isn_s$. ∎

Invariant 7.5

1. $isn_c \neq \mathtt{nil}$ if and only if $mode_c \neq \mathtt{closed}$.

2. $isn_s^c \neq \mathtt{nil}$ if and only if $mode_c \notin \{\mathtt{closed}, \mathtt{syn\text{-}sent}\}$.

3. $isn_s \neq \mathtt{nil} \vee isn_c^s \neq \mathtt{nil}$ if and only if $mode_s \notin \{\mathtt{closed}, \mathtt{listen}\}$. ∎

Invariant 7.6

1. If $isn_c^s \neq \mathtt{nil}$ then $isn_c^s \leq sn_c$.

2. If $isn_c^s \neq \mathtt{nil}$ then $isn_c^s < ack_s$.

3. If $isn_s^c \neq \mathtt{nil}$ then $isn_s^c \leq sn_s$.

4. If $isn_s^c \neq \mathtt{nil}$ then $isn_s^c < ack_c$.

5. If $isn_c \neq \mathtt{nil}$ then $isn_c \leq sn_c$.

6. If $isn_s \neq \mathtt{nil}$ then $isn_s \leq sn_s$. ∎

Invariant 7.7

If $mode_s =$ `syn-rcvd` then $ack_s = isn_c^s + 1$. ∎

Invariant 7.8

If $(i, j) \in assoc$ then $i \leq sn_c \wedge j \leq sn_s$. ∎

Invariant 7.9

1. If $isn_c \neq \mathtt{nil} \wedge choose\text{-}isn_c = \mathtt{true}$ then $isn_c \neq isn_c^s$.

2. If $isn_s \neq \mathtt{nil} \wedge choose\text{-}isn_s = \mathtt{true}$ then $isn_s \neq isn_s^c$. ∎

Invariant 7.10

1. If $mode_c =$ `syn-sent` then $sn_c = isn_c$.

2. If $mode_c =$ `syn-rcvd` then $sn_s = isn_s$. ∎

Invariant 7.11

1. If $choose\text{-}isn_c \wedge isn_c = i$ then $\forall$ SYN segments $p \in in\text{-}transit_{cs}$, $sn(p) < i \wedge$ $\forall$ SYN segments $q \in in\text{-}transit_{sc}$, $ack(q) < i + 1$.

2. If $choose\text{-}isn_s \wedge isn_s = i$ then $\forall$ SYN segments $p \in in\text{-}transit_{sc}$, $sn(p) < j \wedge \forall$ segments $q \in in\text{-}transit_{cs}$, $ack(q) < i + 1$. ■

Invariant 7.12

1. For all $i \in \mathbb{N} \cup \{\texttt{nil}\}$, $(i, \texttt{nil}) \notin estb\text{-}pairs$.

2. For all $j \in \mathbb{N} \cup \{\texttt{nil}\}$, $(\texttt{nil}, j) \notin estb\text{-}pairs$.

3. For all $i \in \mathbb{N} \cup \{\texttt{nil}\}$, $(i, \texttt{nil}) \notin assoc$.

4. For all $j \in \mathbb{N} \cup \{\texttt{nil}\}$, $(\texttt{nil}, j) \notin assoc$. ■

Invariant 7.13 is directly used to in the simulation proof. It states that if a host has started the close phase (indicated by its mode), it must have received the signal to close from the user ($rcvd\text{-}close_c$ or $rcvd\text{-}close_s$ is `true`), and it must have sent all the data it received from the user (the send buffers are empty).

Invariant 7.13

1. If $mode_c \in \{\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$ then $send\text{-}buf_c = \epsilon \wedge rcvd\text{-}close_c = \texttt{true}$.

2. If $mode_s \in \{\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$ then $send\text{-}buf_s = \epsilon \wedge rcvd\text{-}close_s = \texttt{true}$. ■

Invariant 7.14, is also used directly in the simulation proof. It states that before the server gets to a synchronized state, it does not accept any messages, so its receive buffer is empty.

Invariant 7.14

If $mode_s \in \{\texttt{listen}, \texttt{syn-rcvd}\}$ then $rcv\text{-}buf_s = \epsilon$. ■

The next invariant that is directly referred to in the simulation proof is Invariant 7.30. The invariants up to that one are needed for its proof and/or the proof of subsequent invariants. Invariant 7.15 is about the three-way handshake protocol. It states that if the client has sent the segment for the final leg of the protocol ($ack(p) > sn_s$), and neither it nor the server closed since the first segment for the protocol was sent ($isn_c = isn^s_c$), then the client cannot be in mode `syn-sent`.

Invariant 7.15

If $isn_c = isn^s_c$ and there exists $p \in \mathit{in\text{-}transit}_{cs}$ such that $ack(p) > sn_s$ then $mode_c \neq$ `syn-sent`. ■

Parts one and two of the next invariant states that before the client or server gets to a synchronized state, there initial sequence number is not part of an association pair, and Parts three and four states that when the client and server first choose initial sequence numbers, the number is not part of a pair in the set *estb-pairs*.

Invariant 7.16

1. If $mode_s =$ `syn-rcvd` then for all i, $(i, isn_s) \notin assoc$.

2. If $mode_c =$ `syn-sent` then for all j, $(isn_c, j) \notin assoc$.

3. If $mode_s =$ `syn-rcvd` $\wedge\ \mathit{choose\text{-}isn}_s$ then for all i, $(i, isn_s) \notin \mathit{estb\text{-}pairs}$.

4. If $mode_c =$ `syn-sent` then for all j, $(isn_c, j) \notin \mathit{estb\text{-}pairs}$. ■

Invariant 7.17 states that the client's initial sequence number becomes part of a pair in *estb-pairs* if and only if the client is in a synchronized mode, unless it crashed or received a reset.

Invariant 7.17

1. If $mode_c \in \mathit{sync\text{-}states}$ then $(isn_c, isn^c_s) \in \mathit{estb\text{-}pairs}$

2. If $(isn_c, isn^c_s) \in \mathit{estb\text{-}pairs} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ then $mode_c \in \mathit{sync\text{-}states}$. ■

The next invariant states that if the initial sequence number of the client and the initial sequence number of the server form an association pair, then the initial sequence number of the client, and what the client believes to be the initial sequence number of the server isn^c_s are a pair in *estb-pairs*.

Invariant 7.18

If $(isn_c, isn_s) \in assoc \ \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ then $(isn_c, isn_s^c) \in$ *estb-pairs*. ■

Invariants 7.19 and 7.20 are about the open phase of the protocol. Informally speaking, they imply that during this phase of the protocol, the client and server are not out of synch, unless there is a crash or a reset. Invariant 7.19 states that when the client is in mode `syn-sent`, and the server has received the client's initial sequence number, the server cannot yet be in a synchronized mode, and in mode `syn-rcvd`, and Invariant 7.20 states that if the server is in mode `syn-rcvd` it knows the client's initial sequence number and the client knows the server's initial sequence number, then the client must be in a synchronized state.

Invariant 7.19

If $mode_c = \texttt{syn-sent} \wedge isn_c = isn_c^s$ then $mode_s \notin sync\text{-}states$. ■

Invariant 7.20

If $isn_c = isn_c^s \wedge isn_s = isn_s^c \wedge mode_s = \texttt{syn-rcvd} \wedge mode_c \notin \{\texttt{closed}, \texttt{rec}, \texttt{reset}\}$ then $mode_c \in sync\text{-}states$. ■

Invariant 7.21 is needed for the proof of Invariant 7.22 which is in turn needed for the proof of Invariant 7.23. Invariant 7.23 states that whenever the client has an acknowledgment number, it is greater than or equal to the acknowledgment number of any segment on the out going channel of the client. The acknowledgment number of the client gets a non-nil value when the client receives a valid SYN segment from the server. The value is the sequence number plus one of this SYN segment. Invariant 7.22 states that the value of this sequence number is greater than or equal to the acknowledgment number of any segment on the out going channel of the client. The sequence number of this segment was the sequence number of the server in mode `syn-rcvd`, so Invariant 7.21 states that when the server has this sequence number it is greater than or equal to any acknowledgment number on the out going channel of the client.

Invariant 7.21

If $mode_c = \texttt{syn-sent} \wedge mode_s = \texttt{syn-rcvd} \wedge ack_s = sn_c + 1$ then for all segments $p \in in\text{-}transit_{cs}$, $sn_s \geq ack(p)$. ■

Invariant 7.22

If $mode_c = \texttt{syn-sent}$ then for all SYN segments $p \in \mathit{in\text{-}transit}_{sc}$ such that $ack(p) = sn_c + 1$, $sn(p) \geq ack(q)$ for all $q \in \mathit{in\text{-}transit}_{cs}$. ■

Invariant 7.23

If $ack_c \in \mathbb{N}$ then for all $p \in \mathit{in\text{-}transit}_{cs}$, $ack_c \geq ack(p)$. ■

Invariant 7.24 states that under certain conditions the acknowledgment number at the server is always bigger than the acknowledgment number of any segment on the out going channel of the server. This property is almost symmetric to the property expressed in Invariant 7.23. However, there are more conditions in the premise of this invariant because if the server receives an old duplicate SYN segment from the client, it may set its acknowledgment number to a value less than the acknowledgment number of some segments on its out going channel. The conditions in the premise rule out this case.

Invariant 7.24

If $isn_c = isn_c^s \wedge isn_s = isn_s^c \wedge mode_c \in \mathit{sync\text{-}states} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ then for all segments $p \in \mathit{in\text{-}transit}_{sc}$, $ack_s \geq ack(p)$. ■

One important property of the protocol is that if the client's and server's initial sequence numbers are in *assoc* or *estb-pairs* then the client knows the correct initial sequence number of the server and vice-versa. This property is stated as Invariant 7.28. Invariants 7.25, 7.26, and 7.27 are used for the proof of this property. Invariant 7.25 states that if the client is in a synchronized mode and there is a segment from the server on the channel that has data that the client can accept ($sn(p) \geq ack_c$), then either the server is not in mode `syn-rcvd`, or if it is in mode `syn-rcvd`, it is out of synch with the client, so it cannot receive a valid ack from the client because $ack_c < isn_s$. Invariants 7.26, and 7.27 are straightforward. Basically, they state that in the states leading up to when the pair (isn_c, isn_s) pair is added to either *assoc*, or *estb-pairs*, the client and server know the other's initial sequence numbers.

Invariant 7.25

If $mode_c \in \mathit{sync\text{-}states} \wedge mode_s \notin \{\texttt{closed}, \texttt{rec}, \texttt{reset}\} \wedge isn_c = isn_c^s$ and there exists a non-SYN segment $p \in \mathit{in\text{-}transit}_{sc}$ such that $sn(p) \geq ack_c$, then $mode_s \neq \texttt{syn-rcvd} \vee ack_c < isn_s$.

■

Invariant 7.26

If $mode_s = \texttt{syn-rcvd} \wedge isn_c = isn_c^s \wedge ack_c = isn_s + 1$ then $isn_s = isn_s^c$. ■

Invariant 7.27

If $mode_s = \texttt{syn-rcvd} \wedge isn_c = isn_c^s$ and there exists a non-SYN segment $p \in \mathit{in\text{-}transit}_{cs}$ such that $ack(p) = isn_s + 1$ then $isn_s = isn_s^c$. ■

Invariant 7.28

1. If $(isn_c, isn_s) \in assoc$ then $isn_s^c = isn_s \wedge isn_c^s = isn_c$.

2. If $(isn_c, isn_s) \in \mathit{estb\text{-}pairs}$ then $isn_s^c = isn_s \wedge isn_c^s = isn_c$. ■

Invariant 7.29 is very similar to Invariant 7.24. It gives different conditions under which the acknowledgment number at the server is greater than or equal to the acknowledgment number of any out going segments.

Invariant 7.29

If $(isn_c, isn_s) \in assoc \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ then for all segments $p \in \mathit{in\text{-}transit}_{sc}$, $ack_s \geq ack(p)$. ■

Invariant 7.30 is a key one. The conditions in the premise of the invariant are basically the conditions under which a host prepares a message to be sent. Thus, the invariant states that the hosts only prepares new messages if the previous message has been acknowledged ($sn_c < ack_s$ or $sn_s < ack_c$).

Invariant 7.30

1. If $mode_c \in \{\texttt{estb}, \texttt{close-wait}\} \wedge \neg \mathit{ready\text{-}to\text{-}send}_c \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in \mathit{estb\text{-}pairs} \wedge isn_c^s = isn_c$ then $sn_c < ack_s$.

2. If $mode_s \in \{\texttt{estb}, \texttt{close-wait}\} \wedge \neg \mathit{ready\text{-}to\text{-}send}_s \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in assoc$ then $sn_s < ack_c$. ■

Invariant 7.33 is another key invariant. It expresses an essential correctness property of the protocol. It states that an initial sequence number from the client can only be paired with a unique initial sequence number from the server, and vice-versa. Invariants 7.31 and 7.32 are used for the proof of this invariant.

Invariant 7.31

1. If $(i, isn_s) \in assoc$ then $isn_c^s = i$.

2. If $(isn_c^s, j) \in assoc \wedge mode_s \in sync\text{-}states$ then $isn_s = j$. ∎

Invariant 7.32

If $(isn_c^s, j) \in assoc \wedge isn_s \neq j \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ then $mode_s = \texttt{syn-rcvd}$. ∎

Invariant 7.33

1. If $(h, j) \in assoc \wedge (i, j) \in assoc$ then $h = i$.

2. If $(i, j) \in assoc \wedge (i, k) \in assoc$ then $j = k$. ∎

Invariant 7.34 is also used directly in the simulation proof. It states that if the client is in a synchronized state and the initial sequence number of the server is part of a pair in the set *estb-pairs*, then the other part of the pair must be the initial sequence number of the client.

Invariant 7.34

If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states \ \wedge mode_c \in sync\text{-}states$ and there exists i such that $(i, isn_s) \in estb\text{-}pairs$ then $i = isn_c$. ∎

Invariant 7.36 is also important, it says that if the message at a host is not `null`, and there is a segment with the same sequence number as the host, then the segment must have the same message as the host. Invariant 7.37 is another key invariant that express an essential correctness property. It states that if two segments on the same channel have the same sequence number, it the messages on the segments are not `null`, then they must have the same message. Invariant 7.35 is a preliminary step in the proof of Invariant 7.37. It states that if the message at the host is different from the message on a segment, then the sequence number of the segment is strictly less than the current sequence number of the host.

Invariant 7.35

1. If there exists $p \in in\text{-}transit_{cs}$ such that $msg(p) \neq msg_c$ then $sn(p) < sn_c \ \vee msg_c = \texttt{null}$.

2. If there exists $p \in \mathit{in\text{-}transit}_{sc}$ such that $msg(p) \neq msg_s$ then $sn(p) < sn_s \ \vee msg_s =$ `null`. ■

Invariant 7.36

1. If $msg_c \neq$ `null` and there exists $p \in \mathit{in\text{-}transit}_{cs}$ such that $sn(p) = sn_c$ then $msg(p) = msg_c$.

2. If $msg_s \neq$ `null` and there exists $p \in \mathit{in\text{-}transit}_{sc}$ such that $sn(p) = sn_s$ then $msg(p) = msg_s$. ■

Invariant 7.37

1. If there exists segments p and q on $\mathit{in\text{-}transit}_{cs}$ such that $sn(p) = sn(q) \ \wedge msg(p) \neq$ `null` $\wedge \ msg(q) \neq$ `null` then $msg(p) = msg(q)$.

2. If there exists segments p and q on $\mathit{in\text{-}transit}_{sc}$ such that $sn(p) = sn(q) \ \wedge msg(p) \neq$ `null` $\wedge \ msg(q) \neq$ `null` then $msg(p) = msg(q)$. ■

Invariant 7.38 states a property that is easy to see and prove. This property is used directly in the simulation proof.

Invariant 7.38

If $mode_s \in \mathit{sync\text{-}states}$ then $(isn_c^s, isn_s) \in assoc$. ■

Invariant 7.47 states a similar property about the client. It states that when the client is in a mode that indicates it has received a FIN segment from the server, then its initial sequence number is part of an association pair. However, to prove the property for the client side requires a few more steps. When the server first gets to a synchronized mode, its initial sequence number is paired with isn_c^s and added to $assoc$. In order to prove that the client's initial sequence number also becomes a part of an association pair, we have to prove that when the pair is added to $assoc$, that $isn_c^s = isn_s$. This property is stated in Invariant 7.44. To prove this property, we examined properties that are true if $isn_c^s \neq isn_s$. The main property that we show when $isn_c^s \neq isn_s$ is stated in Invariant 7.41. This invariant states that if the client is in a synchronized state and the server is in mode `syn-rcvd`, and if $isn_c^s \neq isn_s$, then there are no segments on the channel that can cause the server to go to a synchronized mode and to add a pair to $assoc$. To prove Invariant 7.41 we use Invariant 7.40 which states that the acknowledgment number of the client is less than $sn_s + 1$ under the same conditions. To prove Invariant 7.40 we use Invariant 7.39 which states that the segment that causes the client to go to a synchronized mode, has sequence number (on which the acknowledgment number of the client is based) that is strictly less than the sequence number of the server when $isn_c \neq isn_c^s$.

Invariant 7.39

If $mode_c = \texttt{syn-sent} \wedge mode_s = \texttt{syn-rcvd} \wedge isn_c \neq isn_c^s$ then for all SYN segments $p \in$ *in-transit*$_{sc}$ such that $ack(p) = sn_c + 1$, $sn(p) < sn_s$. ■

Invariant 7.40

If $mode_c \in$ *sync-states* $\wedge\ mode_s = \texttt{syn-rcvd} \wedge isn_c \neq isn_c^s$ then $ack_c < sn_s + 1$. ■

Invariant 7.41

If $mode_c \in$ *sync-states* $\wedge\ mode_s = \texttt{syn-rcvd} \wedge isn_c \neq isn_c^s$ then for all segments $p \in$ *in-transit*$_{cs}$, $ack(p) < sn_s + 1$. ■

Invariant 7.43 is also used in the proof of Invariant 7.44. It states that when the client first becomes established for an incarnation, the server is not yet in a synchronized mode. To prove Invariant 7.43 we use Invariant 7.42 which states if there is a segment on the incoming channel of the server that can cause it to go to a synchronized mode, then the client cannot be in mode `syn-sent`, because it needs to be in a synchronized mode in order to send this segment, or if the client is in mode `syn-sent`, then it must have closed and reopened, so there are no segments from the server that can acknowledge the new sequence number of the client.

Invariant 7.42

If $mode_s = \texttt{syn-rcvd}$ and there exists $p \in$ *in-transit*$_{cs}$ such that $ack(p) = sn_s + 1$, then $mode_c \neq \texttt{syn-sent}$ or for all SYN segments $q \in$ *in-transit*$_{sc}$, $ack(q) \neq sn_c + 1$. ■

Invariant 7.43

If $mode_c = \texttt{syn-sent}$ and there exists SYN segment $p \in$ *in-transit*$_{sc}$ such that $ack(p) = sn_c + 1$, then $mode_s \notin$ *sync-states* and for all $i \in \mathbb{N}$ $(i, isn_s) \notin$ *assoc*. ■

Invariant 7.44

If $mode_c \in$ *sync-states* $\wedge\ (isn_c^s, isn_s) \in$ *assoc* then $isn_c = isn_c^s$. ■

Once we have Invariant 7.44 we can use it along with Invariant 7.45 and some previously defined invariants to prove Invariant 7.46 which is the invariant directly used in the proof of Invariant 7.47. Invariant 7.46 states that if there is a segment on the incoming channel of the client that can cause it to be in one of the modes in the premise of Invariant 7.47, then the initial sequence number of the client is already part of an association pair. Invariant 7.45 says when the client first gets to a synchronized mode, there are no additional segments on the channel that can cause it to change modes.

Invariant 7.45

If $mode_c = \texttt{syn-sent}$ and there exists a SYN segment $p \in \textit{in-transit}_{sc}$ such that $ack(p) = sn_c + 1$ then for all non-SYN segments $q \in \textit{in-transit}_{sc}$, $sn(q) < sn(p) + 1$. ■

Invariant 7.46

If $mode_c \in \textit{sync-states}$ and there exists a non-SYN segment $p \in \textit{in-transit}_{sc}$ such that $sn(p) \geq ack_c$ then there exists j such that $(isn_c, j) \in assoc$. ■

Invariant 7.47

If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ then $\exists\ j$ such that $(isn_c, j) \in assoc$. ■

The next invariant that is used directly in the simulation proof is Invariant 7.52. It states that if a host is in a mode that indicates it has received a FIN segment, and its initial sequence number is paired with the other host's initial sequence number, then that other host must be in a mode that indicates that it sent the FIN segment. That is, if a host accepts a FIN segment, it must be a legitimate FIN segment for the current incarnation of the connection. To prove Invariant 7.52, we use Invariants 7.49 and 7.51. Invariant 7.49 states that before the server gets to a synchronized state, the client could not have already received a legitimate FIN segment, and Invariant 7.51 states that when there is a legitimate FIN segment from the server on the way to the client, the server must be in a mode that indicates it sent this segment. To prove Invariant 7.49 we use Invariant 7.48 which says essentially the same thing as Invariant 7.49, but the condition in the premise about the state from which the client sends the segment mention in the premise of Invariant 7.49. To prove Invariant 7.51 we use Invariant 7.50 which says if there is a legitimate FIN segment from the client to the server before the server gets to a synchronized mode, the client must be in a mode that indicates that it sent a FIN segment.

Invariant 7.48

If $mode_s = \texttt{syn-rcvd} \wedge isn_c = isn_c^s \wedge ack_c = isn_s + 1$ then $mode_c \notin \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$. ■

Invariant 7.49

If $mode_s = \texttt{syn-rcvd} \wedge \ isn_c = isn_c^s$ and there exists a non-SYN segment $p \in \textit{in-transit}_{cs}$ such that $ack(p) = sn_s + 1$ then $mode_c \notin \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$.
■

Invariant 7.50

If $mode_s = \texttt{syn-rcvd} \wedge mode_c \notin \{\texttt{closed}, \texttt{rec}, \texttt{reset}\}$ and there exists a non-SYN segment $p \in \textit{in-transit}_{cs}$ such that $ack(p) = sn_s + 1$ and there exists a FIN segment $q \in \textit{in-transit}_{cs}$ such that $(sn(q) \geq \max(ack_s, sn(p)+1) \vee (p = q \wedge sn(q) \geq ack_s))$ then $mode_c \in \{\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$. ■

Invariant 7.51

1. If $mode_c \in \textit{sync-states} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in \textit{assoc}$ and there exists a FIN segment $p \in \textit{in-transit}_{sc}$ such that $sn(p) \geq ack_c$ then $mode_s \in \{\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$.

2. If $mode_s \in \textit{sync-states} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in \textit{estb-pairs} \wedge isn_c = isn_c^s$ and there exists a FIN segment $p \in \textit{in-transit}_{cs}$ such that $sn(p) \geq ack_s$ then $mode_c \in \{\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$. ■

Invariant 7.52

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in \textit{assoc}$ then $mode_s \in \{\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in \textit{estb-pairs} \wedge isn_c = isn_c^s$ then $mode_c \in \{\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$. ■

Invariant 7.53 also expresses a key correctness property. It states that when a host receives a segment from which it may accept data ($sn(p) \geq ack_c$ or $sn(p) \geq ack_s$), then the sender has not changed its sequence number from the time it sent this segment. Another way to state the property expressed by the invariant is: sequence numbers do not get changed until the data sent with that sequence number is acknowledged.

Invariant 7.53

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup \textit{sync-states} \wedge mode_c \in \{\texttt{rec}, \texttt{reset}\} \cup \textit{sync-states} \wedge isn_c = isn_c^s \wedge isn_s = isn_s^c$ and there exists $p \in \textit{in-transit}_{cs}$ such that $sn(p) \geq ack_s$, then $sn_c = sn(p)$.

2. If $mode_c \in \textit{sync-states} \wedge (isn_c, isn_s) \in assoc$ and there exists $p \in \textit{in-transit}_{sc}$ such that $sn(p) \geq ack_c$, then $sn_s = sn(p)$. ■

Invariants 7.54, 7.55, and 7.56 state an important correctness property. They state that segments that cause the the value of the message variable on the segment to be added to the receive buffer, contains valid messages. That is, they contain messages that are not `null`.

Invariant 7.54

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup \textit{sync-states} \;\wedge mode_c \in \textit{sync-states} \;\wedge\; (\textit{ready-to-send}_c \vee \textit{send-fin}_c) \wedge\; (isn_c, isn_s) \in \textit{estb-pairs} \;\wedge isn_c^s = isn_c \;\wedge\; ((sn_c = ack_s \wedge \neg(\textit{rcvd-close}_c \wedge \textit{send-buf}_c = \epsilon)) \vee sn_c = ack_s + 1)$ then $msg_c \neq \texttt{null}$.

2. If $mode_c \in \textit{sync-states} \wedge (isn_c, isn_s) \in assoc \;\wedge\; (\textit{ready-to-send}_s \vee \textit{send-fin}_s) \wedge\; ((sn_s = ack_c) \wedge \neg(\textit{rcvd-close}_s \wedge \textit{send-buf}_s \;= \epsilon)) \vee (sn_s = ack_c + 1)$ then $msg_s \neq \texttt{null}$. ■

Invariant 7.55

1. If $mode_s \in \textit{sync-states}$ and there exists non-FIN segment $p \in \textit{in-transit}_{cs}$ such that $sn(p) = ack_s$ or a FIN segment $p \in \textit{in-transit}_{cs}$ such that $sn(p) = ack_s + 1$ then $msg(p) \neq \texttt{null}$.

2. If $mode_c \in \textit{sync-states}$ and there exists non-FIN segment $p \in \textit{in-transit}_{sc}$ such that $sn(p) = ack_c$ or a FIN segment $p \in \textit{in-transit}_{sc}$ such that $sn(p) = ack_c + 1$ then $msg(p) \neq \texttt{null}$. ■

Invariant 7.56

If $mode_s = \texttt{syn-rcvd}$ and there exists non-FIN segment $p \in in\text{-}transit_{cs}$ such that $sn(p) = ack_s$ or a FIN segment $p \in in\text{-}transit_{cs}$ such that $sn(p) = ack_s + 1$ and $ack(p) = sn_s + 1$ then $msg(p) \neq \texttt{null}$. ■

The next invariant is also used directly in the simulation proof. It states that when a host is in a mode that indicates that it received a FIN segment, then if the other host has not closed since sending the FIN segment, its sequence number is less than the acknowledgment number of the host that received the FIN segment. The reason for this is that once a host sends a FIN segment it does not send any more data before it closes, so it does not increase its sequence number, and the host that receives the FIN segment sets its acknowledgment number to the sequence number plus one of the FIN segment.

Invariant 7.57

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ $\wedge\ (isn_c, isn_s) \in assoc$ then $sn_s < ack_c$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ $\wedge\ (isn_c, isn_s) \in estb\text{-}pairs\ \wedge\ isn_c = isn_c^s$ then $sn_c < ack_s$. ■

When a host closes normally in TCP, it closes either from mode `last-ack`, or from mode `timed-wait` after wait for a period of 2μ. Invariant 7.59 states that when a host closes from mode `last-ack` its receive buffer is empty, and Invariant 7.61 says the buffer is empty if the close is from `timed-wait` state. That is, hosts pass all the data to the user before they close under normal conditions. To prove Invariant 7.59 we use Invariant 7.58 which states that when the host is in a mode that indicates that it has received a FIN segment, either a flag ($push\text{-}data_c$ or $push\text{-}data_s$) is set that forces the host to pass all the data to the user before it does anything else, or the receive buffer is empty. In the proof of Invariant 7.61 we use Invariant 7.60. This invariant states that if a host that sent a FIN and also received a FIN (mode is `closing`) then if it has already sent an acknowledgment for the received FIN segment ($send\text{-}fin\text{-}ack_c$ or $send\text{-}fin\text{-}ack_s$ is `true`), its receive buffer must already be empty.

These conditions (mode is $\texttt{closing}$ and $\textit{send-fin-ack}_c$ or $\textit{send-fin-ack}_s$ is $\texttt{true}$) must be true before a host starts $\texttt{timed-wait}$ state.

Invariant 7.58

1. If $\textit{mode}_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{timed-wait}\} \wedge \textit{mode}_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (\textit{isn}_c, \textit{isn}_s) \in \textit{assoc}$ then $\textit{push-data}_c = \texttt{true} \vee \textit{rcv-buf}_c = \epsilon$.

2. If If $\textit{mode}_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{timed-wait}\} \wedge \textit{mode}_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (\textit{isn}_c, \textit{isn}_s) \in \textit{estb-pairs} \wedge \textit{isn}_c = \textit{isn}_c^s$ then $\textit{push-data}_s = \texttt{true} \vee \textit{rcv-buf}_s = \epsilon$. ■

Invariant 7.59

1. If $\textit{mode}_c = \texttt{last-ack} \wedge \textit{mode}_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (\textit{isn}_c, \textit{isn}_s) \in \textit{assoc}$ then $\textit{rcv-buf}_c = \epsilon$.

2. If $\textit{mode}_s = \texttt{last-ack} \wedge \textit{mode}_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (\textit{isn}_c, \textit{isn}_s) \in \textit{estb-pairs} \wedge \textit{isn}_c = \textit{isn}_c^s$ then $\textit{rcv-buf}_s = \epsilon$. ■

Invariant 7.60

1. If $\textit{mode}_c = \texttt{closing} \wedge \textit{send-fin-ack}_c = \texttt{true}$ then $\textit{rcv-buf}_c = \epsilon$.

2. If $\textit{mode}_s = \texttt{closing} \wedge \textit{send-fin-ack}_c = \texttt{true}$ then $\textit{rcv-buf}_s = \epsilon$. ■

Invariant 7.61

1. If $\textit{mode}_c = \texttt{timed-wait} \wedge \textit{first}(\textit{t-out}_c) \in \texttt{T}$ then $\textit{rcv-buf}_c = \epsilon$.

2. If $\textit{mode}_s = \texttt{timed-wait} \wedge \textit{first}(\textit{t-out}_s) \in \texttt{T}$ then $\textit{rcv-buf}_s = \epsilon$. ■

The remaining invariants are about situations where the hosts have formed an incarnation or are about to form one, but one of the host may have closed since the incarnation was formed. When it is the server that may have closed, it is indicated in the invariants as the property $\textit{mode}_c \in \textit{sync-states}$ and there exists j such that $(\textit{isn}_c, j) \in \textit{assoc}$, and when it is the client that may have closed, it is indicated by the property $\textit{mode}_s \in \{\texttt{syn-rcvd}\} \cup \textit{sync-states}$ and there exists i, such that $i = \textit{isn}_c^s \wedge (i, \textit{isn}_s) \in \textit{estb-pairs}$.

Invariant 7.62 is used directly in the simulation proof. The property it expresses is important for the $\textit{p-pair}_c$ and $\textit{p-pair}_s$ derived variables. The invariant states that when a host receives a segment that may have acceptable data ($\textit{sn}(p) \geq \textit{ack}_c$ or $\textit{sn}(p) \geq \textit{ack}_s$), then all other segments q on the channel have $\textit{sn}(q) \leq \textit{sn}(p)$. This means that if the message was a part of a possible pair, the set becomes empty after this message is received because when the segment is received the acknowledgment number of the receiving host is set to $sn(p)+1$. This invariant and Invariant 7.64 also gives the property that in any state there is at most one element in the possible pairs sets. Invariant 7.64 states that there cannot be segments on the channel at the same time that have sequence number equal to the acknowledgment number of the receiving host plus one, and also segments that have sequence number equal to the acknowledgment number of the receiving host. Invariant 7.63 is used in the proof of Invariant 7.64. It states that if the actual sequence number of the sending host is equal to the acknowledgment number of the receiving host plus, then there are no segments on the outgoing channel of the sender that has sequence number equal to the acknowledgment number of the receiving host.

Invariant 7.62

1. If $mode_c \in$ *sync-states* and there exists j such that $(isn_c, j) \in assoc$ and there exists a non-SYN segment $p \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, then for all non-SYN segments $q \in in\text{-}transit_{sc}$ $sn(q) \leq sn(p)$.

2. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states$ and there exists i, such that $i = isn_c^s \wedge (i, isn_s) \in$ *estb-pairs* and there exists a non-SYN segment $p \in in\text{-}transit_{cs}$ such that $sn(p) \geq ack_s$, then for all non-SYN segments $q \in in\text{-}transit_{cs}$ $sn(q) \leq sn(p)$. ■

Invariant 7.63

1. If $mode_c \in \{\texttt{rec}, \texttt{reset}\} \cup sync\text{-}states \wedge mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states \wedge isn_c = isn_c^s \wedge (isn_c, isn_s) \in estb\text{-}pairs \wedge sn_c = ack_s + 1$ then for all non-SYN segments $p \in in\text{-}transit_{cs}$, $sn(p) \neq ack_s$.

2. If $mode_c \in sync\text{-}states \wedge (isn_c, isn_s) \in assoc \wedge sn_s = ack_c + 1$ then for all non-SYN segments $p \in in\text{-}transit_{sc}$, $sn(p) \neq ack_c$. ■

Invariant 7.64

1. If $mode_c \in$ *sync-states* and there exists j such that $(isn_c, j) \in assoc$ and there exists a non-SYN segment $p \in in\text{-}transit_{sc}$ such that $sn(p) = ack_c + 1$, then for all non-SYN segments $q \in in\text{-}transit_{sc}$, $sn(q) \neq ack_c$.

2. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states$ and there exists i, such that $i = isn_c^s \wedge (i, isn_s) \in estb\text{-}pairs$ and there exists a non-SYN segment $p \in in\text{-}transit_{cs}$ such that $sn(p) = ack_s + 1$, then for all non-SYN segments $q \in in\text{-}transit_{cs}$, $sn(q) \neq ack_s$. ■

Invariant 7.65 is similar to Invariant 7.57, and serves a similar purpose. However, in this invariant, since the sending host may have closed, the property expressed by the invariant compares the sequence number of segments as opposed to the actual sequence numbers.

Invariant 7.65

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists j such that $(isn_c, j) \in assoc$ then for all non-SYN segments $p \in in\text{-}transit_{sc}$, $sn(p) < ack_c$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists i, such that $i = isn_c^s \ \wedge \ (i, isn_s) \in estb\text{-}pairs$ then for all non-SYN segments $p \in in\text{-}transit_{cs}$, $sn(p) < ack_c$. ■

Invariants 7.66 and 7.67 are similar to Invariants 7.58 and 7.59 respectively. The difference being that the properties are expressed for the situation where one of the hosts might have closed after the connection is formed.

Invariant 7.66

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{timed-wait}\}$ and there exists j such that $(isn_c, j) \in assoc$ then $push\text{-}data_c = \texttt{true} \ \vee \ rcv\text{-}buf_c = \epsilon$.

2. If If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{timed-wait}\}$ there exists i such that $(i, isn_s) \in estb\text{-}pairs \ \wedge \ isn_c = isn_c^s$ then $push\text{-}data_s = \texttt{true} \ \vee \ rcv\text{-}buf_s = \epsilon$. ■

Invariant 7.67

1. If $mode_c = \texttt{last-ack}$ and there exists j such that $(isn_c, j) \in assoc$ then $rcv\text{-}buf_c = \epsilon$.

2. If $mode_s = \texttt{last-ack}$ and there exists i, such that $i = isn_c^s \ \wedge \ (i, isn_s) \in estb\text{-}pairs$ then $rcv\text{-}buf_s = \epsilon$. ■

The final significant invariant is Invariant 7.70. It is used directly in the simulation proof, and it states that if a host is in mode that indicates that it received a FIN segment, then the other host must either have the flag set that indicates it received a close signal from its user, or if the flag is not set to true, it must be because the host closed after sending the FIN segment. Invariants 7.68 and 7.69 are used in the proof of this invariant. They are similar to Invariants 7.50 and 7.51 respectively.

Invariant 7.68

If $mode_s = \texttt{syn-rcvd}$ and there exists a non-SYN segment $p \in \mathit{in\text{-}transit}_{cs}$ such that $ack(p) = sn_s + 1$ and there exists a FIN segment $q \in \mathit{in\text{-}transit}_{cs}$ such that $(sn(q) \geq \max(ack_s, sn(p)+1) \vee (p = q \wedge sn(q) \geq ack_s))$ then $\mathit{rcvd\text{-}close}_c = \texttt{true} \vee isn_c \neq isn_c^s$. ■

Invariant 7.69

1. If $mode_c \in \mathit{sync\text{-}states}$ and there exists j such that $(isn_c, j) \in \mathit{assoc}$ and there exists a FIN segment $p \in \mathit{in\text{-}transit}_{sc}$ such that $sn(p) \geq ack_c$ then $\mathit{rcvd\text{-}close}_s = \texttt{true} \vee isn_s \neq j$.

2. If $mode_s \in \mathit{sync\text{-}states}$ and there exists i such that $(i, isn_s) \in \mathit{estb\text{-}pairs} \wedge i = isn_c^s$ and there exists a FIN segment $p \in \mathit{in\text{-}transit}_{cs}$ such that $sn(p) \geq ack_s$ then $\mathit{rcvd\text{-}close}_c = \texttt{true} \vee isn_c \neq i$. ■

Invariant 7.70

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists j such that $(isn_c, j) \in \mathit{assoc}$ then $\mathit{rcvd\text{-}close}_s = \texttt{true} \vee isn_s \neq j$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists i such that $(i, isn_s) \in \mathit{estb\text{-}pairs} \wedge i = isn_c^s$ then $\mathit{rcvd\text{-}close}_c = \texttt{true} \vee isn_c \neq i$. ■

The conjunction of all the above invariants is itself an invariant, and we call this Invariant I_T.

7.3 The simulation proof

In this section we define a mapping from states of $\mathcal{TCP}^h$ to states of D^p, and then prove that it is a timed refinement mapping with respect to Invariants I_T and I_D.

7.3.1 The refinement mapping

We define a function R_{TD} from *states($\mathcal{TCP}^h$)* to *states(D^p)*. In the definition, when we write, for example, "($\mathit{send\text{-}buf}_c \times \mathtt{ok}$)", we mean the element of *(Msg $\times$ Flag)** obtained from $\mathit{send\text{-}buf}_c$ by pairing every message with `ok`. If $\mathit{send\text{-}buf}_c$ is undefined or empty then "($\mathit{send\text{-}buf}_c \times \mathtt{ok}$)" is the empty string.

Definition 7.1 (Refinement Mapping from $\mathcal{TCP}^h$ to D^p)
For our mapping the *CID* and *SID* are instantiated by the set of non-negative integers. If $s \in$ *states($\mathcal{TCP}^h$)* then define R_{TD} to be the state $u \in$ *states(D^p)* such that:

1. $u.now = s.now$

2. $u.\mathit{choose\text{-}sid} = (s.mode_s = \mathtt{listen})$

3. $u.rec_c = (s.mode_c = \mathtt{rec})$
   $u.rec_s = (s.mode_s = \mathtt{rec})$

4. $u.abrt_c = (s.mode_c = \mathtt{reset})$
   $u.abrt_s = (s.mode_s = \mathtt{reset})$

5. $u.\mathit{used\text{-}id}_c = s.\mathit{used\text{-}id}_c$
   $u.\mathit{used\text{-}id}_s = s.\mathit{used\text{-}id}_s$

6. $$\begin{aligned} u.id_c &= s.isn_c \\ u.id_s &= s.isn_s \end{aligned}$$

7. $u.assoc = s.assoc$

8. $$\begin{aligned} u.mode_c &= \mathtt{active} && \text{if } s.\mathit{rcvd\text{-}close}_c = \mathtt{false} \\ &= \mathtt{inactive} && \text{if } s.\mathit{rcvd\text{-}close}_c = \mathtt{true} \vee mode_c = \mathtt{closed} \\ u.mode_s &= \mathtt{active} && \text{if } s.\mathit{rcvd\text{-}close}_s = \mathtt{false} \\ &= \mathtt{inactive} && \text{if } s.\mathit{rcvd\text{-}close}_s = \mathtt{true} \vee mode_s = \mathtt{closed} \end{aligned}$$

9. $$\begin{aligned} u.\mathit{q\text{-}stat}_{cs}(i) &= \mathtt{live} && \text{if } (s.isn_c = i \ \wedge \ \forall\ j\ (i,j) \notin s.\mathit{estb\text{-}pairs}) \ \vee ((i, s.isn_s) \in \\ &&& s.\mathit{estb\text{-}pairs} \ \wedge \ s.isn_c^s = i \ \wedge s.mode_s \notin \{\mathtt{rec}, \mathtt{reset}\}) \\ &= \mathtt{dead} && \text{otherwise} \\ u.\mathit{q\text{-}stat}_{sc}(j) &= \mathtt{live} && \text{if } (s.isn_s = j \ \wedge \ \forall\ i, (i,j) \notin s.assoc) \ \vee ((s.isn_c, j) \in s.assoc \wedge \\ &&& s.mode_c \notin \{\mathtt{rec}, \mathtt{reset}\}) \\ &= \mathtt{dead} && \text{otherwise} \end{aligned}$$

10.	$u.queue_{cs}(i)$	$=$	ϵ	if $(s.isn_c \neq i \;\wedge \forall\; j\; (i,j) \notin s.estb\text{-}pairs) \vee ((i,j) \in s.estb\text{-}pairs \;\wedge (s.isn^s_c \neq i \vee s.mode_s \in \{\texttt{rec}, \texttt{reset}\}))$ (A)
		$=$	$(s.send\text{-}buf_c \times \texttt{ok})$	if $s.isn_c = i \wedge \forall\; j\; (i,j) \notin s.estb\text{-}pairs \;\wedge s.mode_c \in \{\texttt{syn-sent}, \texttt{rec}, \texttt{reset}\}$ (B)
		$=$	concatenation of: $\bullet(s.rcv\text{-}buf_s \times \texttt{ok})$ $\bullet\; s.current\text{-}msg_c$ $\bullet(s.send\text{-}buf_c \times \texttt{ok})$	if $s.isn_c = i \;\wedge\; s.mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (i, s.isn_s) \in s.estb\text{-}pairs \;\wedge s.isn^s_c = i \;\wedge s.mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ (C)
		$=$	concatenation of: $\bullet(s.rcv\text{-}buf_s \times \texttt{ok})$ $\bullet(data(s.p\text{-}pair_c)$ $\times\; \texttt{marked})$	if $(s.isn_c \neq i \vee s.mode_c \in \{\texttt{rec}, \texttt{reset}\}) \wedge ((i, s.isn_s) \in s.estb\text{-}pairs \;\wedge s.isn^s_c = i \wedge s.mode_s \notin \{\texttt{rec}, \texttt{reset}\})$ (D)
11.	$u.queue_{sc}(j)$	$=$	ϵ	if $(s.isn_s \neq j \wedge\; \forall\; i, (i,j) \notin s.assoc) \vee ((i,j) \in s.assoc \wedge (s.isn_c \neq i \vee s.mode_c \in \{\texttt{rec}, \texttt{reset}\}))$ (A)
		$=$	$(s.send\text{-}buf_s \times \texttt{ok})$	if $s.isn_s = j \wedge \forall\; i, (i,j) \notin s.assoc \;\wedge s.mode_s \in \{\texttt{syn-rcvd}, \texttt{rec}, \texttt{reset}\}$ (B)
		$=$	concatenation of: $\bullet(s.rcv\text{-}buf_c \times \texttt{ok})$ $\bullet\; s.current\text{-}msg_s$ $\bullet(s.send\text{-}buf_s \times \texttt{ok})$	if $s.isn_s = j \;\wedge\; s.mode_s \notin \{\texttt{rec}, \texttt{reset}\} \;\wedge (s.isn_c, j) \in s.assoc \;\wedge s.mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ (C)
		$=$	concatenation of: $\bullet(s.rcv\text{-}buf_c \times \texttt{ok})$ $\bullet(data(s.p\text{-}pair_s)$ $\times\; \texttt{marked})$	if $(s.isn_s \neq j \vee\; s.mode_s \in \{\texttt{rec}, \texttt{reset}\}) \wedge ((s.isn_c, j) \in s.assoc \wedge s.mode_c \notin \{\texttt{rec}, \texttt{reset}\})$ (D)

We present some intuition behind the mapping. The choosing of *initial sequence numbers* by the client and server in $\mathcal{TCP}^h$, corresponds to the choosing of *ids* by the client and server in D^p. In $\mathcal{TCP}^h$, when the server opens it does not immediately choose an initial sequence number, but chooses one when it receives a SYN segment from the client. Therefore, we map the state where $mode_s$ is `listen` in $\mathcal{TCP}^h$, to the state where *choose-sid* is `true` in the specification. The mapping of *s.now*, $s.used\text{-}id_c$, $s.used\text{-}id_s$, and *s.assoc* are all straightforward. When $s.mode_c = \texttt{rec}$ and $s.mode_s = \texttt{rec}$ obviously correspond to the the states in the specification where $u.rec_c$ and $u.rec_s$ respectively, are true. Similarly, $s.mode_c = \texttt{reset}$ and $s.mode_s = \texttt{reset}$ correspond to the states where $u.abrt_c$ and $u.abrt_s$ respectively are true.

In the specification, a host is `active` when it receives an *open* input, but has not yet received a *close* input for the current incarnation. In $\mathcal{TCP}^h$ when the client or server receives the signal to open $rcvd\text{-}close_c$ or $rcvd\text{-}close_s$ respectively, is initialized to `false`. When the signal to close is received, $rcvd\text{-}close_c$ or $rcvd\text{-}close_s$ is set to `true`. Therefore, $rcvd\text{-}close_c$

or *rcvd-close*$_s$ having the value `false` maps to the respective host in D^p having its mode be `active` in the specification, and *rcvd-close*$_c$ or *rcvd-close*$_s$ having the value `true` or the host being closed corresponds to the respective host having its mode be `inactive` in the specification. We need to map the mode of the host being closed in $\mathcal{TCP}^h$ to the mode of the host being `inactive` in the specification because *rcvd-close*$_c$ and *rcvd-close*$_s$ are undefined when a host is closed.

In $\mathcal{TCP}^h$ there are four variables that correspond to parts of the abstract queue for messages going in a particular direction. For example, messages from the client to the server may be in *send-buf*$_c$, msg_c, *in-transit*$_{cs}$, and *rcv-buf*$_s$. If any of these variables contain a valid message, then the abstract queue to which that variable is mapped must be `live`. For example, when the client opens and assigns isn_c the value i, it may also add a valid message to $sbuf_c$, so the abstract queue, $queue_{cs}(i)$, becomes `live`. This abstract queue remains `live` as long as the client has $isn_c = i$. Even if client crashes, receives a reset, or closes, $queue_{cs}(i)$ remains `live` if $((i, isn_s) \in estb\text{-}pairs)$ and $(isn_c^s = i)$ and $(mode_c \notin \{\texttt{rec}, \texttt{reset}\})$. The queue, remains `live` in this situation because there might still be a messages from the client in *in-transit*$_{cs}$ and *rcv-buf*$_s$ that the server may deliver to its user. For this case we use the condition $(i, isn_s) \in estb\text{-}pairs$ as opposed to $(i, isn_s) \in assoc$ because the client may close while there is a segment on the channel with a valid message, which if it arrives at the server and the server has not crashed or closed, causes the pair (i, isn_s) to be added to *assoc*, and the message to be delivered. In these situations the queue becomes `dead` if the server crashes or closed, because if the server crashes or closes no more data can be received from the corresponding abstract queue. However, in the situation before the client's initial sequence number is paired with an initial sequence number from the server and added to *estb-pairs*, it does not matter if the server has crashed or closed, so the queue is `live`. For the variables that map to an abstract queue, $queue_{sc}(j)$, that take messages from the server to the client, the situation is essentially symmetric, except that $(isn_c, j) \in assoc$ is the condition required for messages to be still valid if the server crashes, resets, or closes after choosing $isn_s = j$.

We break the different states of $\mathcal{TCP}^h$ that maps to states of D^p where a queue is `live` into three different cases for abstract queues in each direction. We also have a fourth case

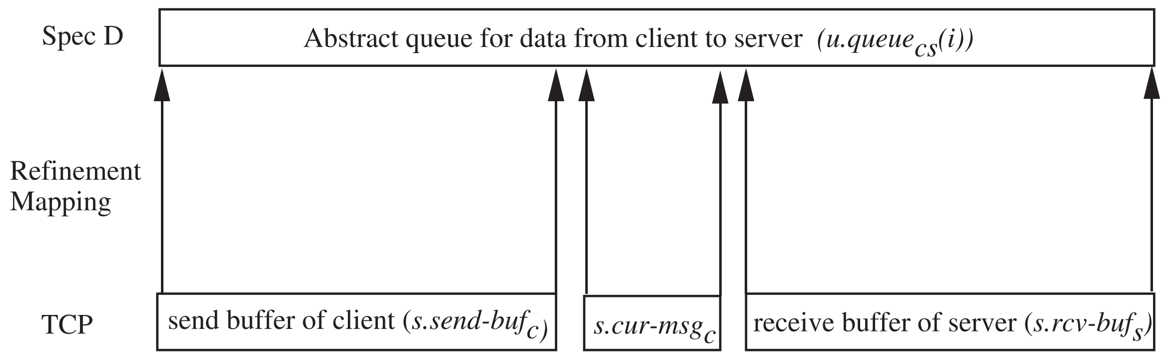

Figure 7-2: *Example of the mapping of variables of TCP to the abstract queue of the specification.*

for queues in each direction, which corresponds to the situation where the abstract queue is dead. We refer to the cases as (A), (B), (C), and (D), for queues in each direction.

For $queue_{cs}(i)$ the first case, (A), is the situation where the states of $\mathcal{TCP}^h$ maps to $q\text{-}stat_{cs}$ being dead. For this situation $queue_{cs}(i)$ is empty.

The second case, (B), is the situation where the client first assigns isn_c to i, and before $mode_c$ gets to `estb`. For this case the queue is just $(s.send\text{-}buf_c \times \mathtt{ok})$ even if the client is in recovery or reset mode.

Case (C) occurs when the client is in a synchronized mode, $((isn_c, isn_s) \in estb\text{-}pairs)$, the server knows the current initial sequence number of the client ($isn_c^s = i$), and the server is not in recovery or reset mode. This case is illustrated in Figure 7-2. This situation, where the client and server are either synchronized or are about to become synchronized, is the typical data transfer state of the protocol . In this situation it is clear that the send buffer of the client should map to a suffix of the abstract queue, and that the receive buffer of the server should map to a prefix of the abstract queue. The tricky part to deal with is the current message being sent, msg_c, because it may have duplicates on the channel and another duplicate in the receive buffer of the server. Our definition of $cur\text{-}msg_c$ handles this situation because while the message is being sent and before it is received by the server, copies on the channel are ignored in the mapping, and $cur\text{-}msg_c$ is msg_c paired with `ok`. However, when the message is received by the client and placed on $rcv\text{-}buf_s$, $cur\text{-}msg_c$ becomes the empty string.

The duplicates on the channel are ignored until there is a crash, reset, or close at the client which brings us to the fourth case, (D), for the mapping to $queue_{cs}(i)$. If there is a crash or the client has received a valid reset but has not yet closed, then $mode_c \in \{\texttt{rec}, \texttt{reset}\}$, This is the first set of states for case (D). If the client closes upon recovery from the crash, or closes for any other reason then $(isn_c \neq i)$ which gives us the second set of states for case (D). In both these situations messages on $send\text{-}buf_c$ are lost and the message in $s.cur\text{-}msg_c$ is either lost or becomes *message(*p-pair$_c$*)*. The server can also still deliver messages from its receive buffer, so the variables that map to the abstract queue in this situation are $(msg(p\text{-}pair_c) \times \texttt{marked})$ and $(s.rcv\text{-}buf_s \times \texttt{ok})$. The message in $p\text{-}pair_c$ is paired with `marked` because it may get dropped.

The four cases for the mapping to queues that take data from the server to the client, $queue_{sc}(j)$, is essentially symmetric, to the mapping to $queue_{cs}(i)$.

Note for the mapping of the queues, that the conditions that determine whether a queue is in a particular group makes the groups mutually disjoint.

7.3.2 Simulation of steps

In this section we prove that the mapping R_{TD} defined in the previous section is indeed a timed refinement mapping from $\mathcal{TCP}^h$ to D^p with respect to I_D and I_T. This claim is stated as the following lemma.

Lemma 7.1
$\mathcal{TCP}^h \leq_R^t D^p$ via R_{TD}.

Proof: We prove that R_{TD} is a timed refinement mapping from $\mathcal{TCP}^h$ to D^p with respect to I_D and I_T by showing that the two cases of Definition 3.11 are satisfied.

Base Case

In the start state s_0 of $\mathcal{TCP}^h$ we have $s_0.mode_c = s_0.mode_s = \texttt{closed}$, $s_0.now = 0$, $s_0.used\text{-}id_c$, $s_0.used\text{-}id_s$, and $s_0.assoc = \emptyset$. It is clear that $R_{\text{TD}}(s_0)$ is the unique start state u_0 of D^p.

Inductive Case

Assume $(s, a, s') \in$ *Steps($\mathcal{TCP}^h$)*. Below we consider cases based on a and for each case we define a finite execution fragment α of S such that $fstate(\alpha) = R_{\text{TD}}(s)$, $lstate(\alpha) = R_{\text{TD}}(s')$, and *t-trace(*$\alpha$*)* $=$ *t-trace(*s, a, s'*)*. For the steps of the proof below we do not include the time of occurrence and last time in the *timed traces* of (s, a, s') or α, so as not to clutter the proof. However, it is clear that since the time-passage steps in D^p are arbitrary, if we show *trace(*α*)* $=$ *trace(*s, a, s'*)* then *t-trace(*α*)* $=$ *t-trace(*s, a, s'*)*. We use u and u' to denote $R_{\mathrm{TD}}(s)$ and $R_{\mathrm{TD}}(s')$ respectively. For symmetric steps we only show the proof correspondence of one of the actions since the proof of correspondence of the other will be symmetric.

a = *send-msg*$_c$*(open, m, close)*.

This step has eight variations depending on whether *open* and *close* are true or false, and whether m is `null` or not. We examine each variation as a separate case.

1. The first variation we examine is *send-msg*$_c$*(true,* `null`*, false)*. For this case, $\alpha = (u, a, u')$. If $s.mode_c =$ `closed` then $s'.mode_c =$ `syn-sent`, $s'.isn_c = s.sn_c{+}1$, $s'.used\text{-}id_c = s.used\text{-}id_c \cup \{s'.isn_c\}$. The corresponding execution fragment α in D^p causes $u'.mode_c$ to be `active`, $u'.id_c$ to be $c \in CID$, $u'.used\text{-}id_c$ to be $u.used\text{-}id_c \cup \{u'.id_c\}$, and $u.q\text{-}stat_{cs}(u'.id_c)$ to be `live`. These states are the correct ones as defined by the mapping R_{TD}.

2. The second variation is *send-msg*$_c$*(false,* `null`*, false)*. For this subcase $\alpha = (u, a, u')$. The action a has no effect on state s nor does it have an effect on state u.

3. The third variation is *send-msg*$_c$*(true, m, false)* where $m \neq$ `null`. For this case $\alpha = (u, a, u')$, and the correspondence for $u'.mode_c$, $u'.id_c$, $u'.q\text{-}stat_{cs}(u'.id_c)$, and $u'.used\text{-}id_c$ is the same as for the first variation. However, this step has the additional effect that m gets concatenated to the end of $s.send\text{-}buf_c$ in $\mathcal{TCP}^h$ if $s.rcvd\text{-}close_c$ is `false`. Since $s'.isn_c \neq$ `nil` and $s'.mode_c \neq$ `rec`, we have case (B) or (C) for $u'.queue_{cs}(u'.id_c)$. If $s.rcvd\text{-}close_c$ is `false`, then in the corresponding state of D^p, $u.mode_c =$ `active`. Therefore, if m gets concatenated to $s.send\text{-}buf_c$ in $\mathcal{TCP}^h$, then α in D^p causes $(m,$ `ok`$)$ to be concatenated to the end of $u.queue_{cs}(u.id_c)$ in D^h, so the resulting states correspond. If m does not get concatenated to $s.send\text{-}buf_c$ because $s.rcvd\text{-}close_c$ is `true`, then $(m,$ `ok`$)$ doe not get concatenated to the end of $u.queue_{cs}(u.id_c)$ in D^h, because $u.mode_c =$ `inactive`. If m does not get concatenated

to $send\text{-}buf_c$ because $mode_c \notin \{\texttt{syn-sent}, \texttt{estb}, \texttt{close-wait}\}$, then Invariant 7.13 tells us that $s.rcvd\text{-}close_c$ is `true`.

4. The fourth variation is $send\text{-}msg_c(false,\ m,\ false)$. Again $\alpha = (u, a, u')$. The only effect this step can have is to to add m to $s.send\text{-}buf_c$. The correspondence is shown as in the previous case.

5. The fifth variation is $send\text{-}msg_c(true,\ m,\ true)$. Again we have $\alpha = (u, a, u')$. This case is the same as that for variation $send\text{-}msg_c(true,\ m,\ false)$, except if $s.mode_c \in \{\texttt{estb}, \texttt{syn-sent}, \texttt{close-wait}\}$, then $s'.rcvd\text{-}close_c = \texttt{true}$. In D^p, α sets $u'.mode_c$ to `inactive`. If $s.mode_c \in \{\texttt{estb}, \texttt{syn-sent}, \texttt{close-wait}\}$ then having $s'.rcvd\text{-}close_c$ set to true corresponds to $u'.mode_c = \texttt{inactive}$, and if $s.mode_c \notin \{\texttt{estb}, \texttt{syn-sent}, \texttt{close-wait}\}$, that corresponds to $u.mode_c = \texttt{inactive}$, so we still get the correct mapping with $u'.mode_c = \texttt{inactive}$.

6. The sixth variation is $send\text{-}msg_c(false,\ m,\ true)$. This case is similar to the previous case, with the difference being it does not have the changes in state that could occur if *open* is true. Thus, $\alpha = (u, a, u')$, and the correspondence is preserved.

7. The seventh variation is $send\text{-}msg_c(true,\ \texttt{null},\ true)$. The case can be broken down into two subcases.

   (a) The first subcase occurs when $s.mode_c \in \{\texttt{closed}, \texttt{syn-sent}\}$ and $s.send\text{-}buf_c$ is empty. After a, $s'.mode_c = \texttt{closed}$ and $s'.rcvd\text{-}close_c = \texttt{true}$. For this subcase, $\alpha = (u,\ send\text{-}msg_c(true,\ \texttt{null},\ true),\ u'',\ reset\text{-}nil_c,\ u')$. The traces are clearly equal, so we need to show that α is enabled in D^p and that it produces the a state that corresponds to s' in our mapping. Since $send\text{-}msg_c(true,\ \texttt{null},\ true)$ is an input action it is always enabled. After this step, $u''.mode_c = \texttt{inactive}$, $u''.id_c$ has a value from *CID*, no association is formed with $u.id_c$ as yet, and $u''.queue_{cs}(u''.id_c)$ is empty. These are precisely the preconditions for enabling the internal action $reset\text{-}nil_c$. After α, $u'.id_c = \texttt{nil}$ and $u'.q\text{-}stat_{cs}(u.id_c) = \texttt{dead}$. This is the correct state for our mapping.

(b) The second subcase occurs for all other states and can be treated like the case five (*send-msg*$_c$*(true, m, true)*), except for the fact that the message buffers do not change. For this subcase $\alpha = (u, a, u')$.

8. The final variation is *send-msg*$_c$*(false,* `null`*, true)*. If $s.mode_c$ = `syn-sent` and $s.send\text{-}buf_c$ is empty, then we have α = *(u, send-msg*$_c$*(false,* `null`*, true), u'', reset-nil*$_c$*, u')*, and the correspondence can be shown as in the previous case. If we are not in this state, then we have $\alpha = (u, a, u')$. The only change that may be caused by this step is if $s.mode_c \in \{$`estb`, `syn-sent`, `close-wait`$\}$, then $s'.rcvd\text{-}close_c$ = `true`. This was one of the possibilities for variation five, and the proof that the states correspond is the same as that case.

a = *passive-open*.

For this step $\alpha = (u, a, u')$. The effect of this step in $\mathcal{TCP}^h$ is to cause $s'.mode_s$ to be `listen` and to initialize TCB_s. In D^p the *passive-open* action causes $u'.mode_s$ = `active` and $u'.choose\text{-}sid$ = `true`, so the mapping between states is preserved.

a = *send-msg*$_s$*(m, close)*.

This step has four variations, each of which we present has a separate case. For all variations a only changes the state of $\mathcal{TCP}^h$ if $s.mode_s \in \{$`listen`, `syn-rcvd`, `estb`, `close-wait`$\}$.

1. The first variation is *send-msg*$_s$(`null`, *true*). For this variation there are two subcases.

   (a) The first subcase occurs when $s.mode_s$ = `listen` and $s.send\text{-}buf_s$ is empty. For this subcase α = *(u, send-msg*$_s$*(ϵ, true), u'', reset-nil*$_s$*, u')*. Clearly the traces are the same, and after a $s'.mode_s$ = `closed` and $s'.isn_s$ = `nil`, and after α, $u'.id_s$ = `nil` and $u'.q\text{-}stat_{sc}(u.id_s)$ = `dead`. Thus, we only need to show that α is enabled in D^p. After *send-msg*$_s$(`null`, *true*), $u''.mode_s$ = `inactive`, $u''.id_s$ has a value from *SID*, no association has been formed with $u''.id_s$ as yet (since $s.mode_s$ = `listen` and for the corresponding state u there is no association formed with the current $u.id_c$ as yet), and $u''.queue_{sc}(id_s)$ is empty. These are precisely the preconditions for the action *reset-nil*$_s$.

(b) The second subcase is for all other states s. The corresponding α in D^p is (u, a, u'). The action a may have an effect on state s only if $s.mode_s \in \{\texttt{syn-rcvd}, \texttt{estb}, \texttt{close-wait}\}$. The effect is to make $s'.rcvd\text{-}close_s$ `true`. After α in D^p $u'.mode_s = \texttt{inactive}$, which gives us the correct correspondence of states.

2. The second variation is $send\text{-}msg_s(\texttt{null}, false)$. For this case $\alpha = (u, a, u')$, and a has no effect on $\mathcal{TCP}^h$ and α has no effect on D^p.

3. The third variation is $send\text{-}msg_s(m,\ true)$ where $m \neq \texttt{null}$. Again we have $\alpha = (u, a, u')$. If $s.mode_s \in \{\texttt{syn-rcvd}, \texttt{estb}, \texttt{close-wait}\} \wedge \neg s.rcvd\text{-}close_s$, then after a, in $\mathcal{TCP}^h$ we have $s'.send\text{-}buf_s = s.\ send\text{-}buf_s$ concatenated with m and $s'.rcvd\text{-}close_s = \texttt{true}$. In the corresponding state of D^p, $u.\text{mode}_s = \texttt{active}$, then after α we have $u'.mode_s = \texttt{inactive}$ and $u'.queue_{sc}(u'.id_s) = u.queue_{sc}(u.id_s)$ concatenated with $(m, \texttt{ok})$. For all other states s of $\mathcal{TCP}^h$, a has no effect because Invariant 7.13 tells us that $s.rcvd\text{-}close_s$ is `true` for these states. Since $s.rcvd\text{-}close_s$ is `true`, the corresponding states u of D^p, $u.mode_s = \texttt{inactive}$. Thus, α has no effect in these states, which gives us the correct correspondence of states.

4. The final variation is $send\text{-}msg_s(m,\ false)$. Again $\alpha = (u, a, u')$. The possible effect of a is to add m to $s.send\text{-}buf_s$. For the corresponding state u, α adds $(m, \texttt{ok})$ to $u.send\text{-}buf_s$. Thus, correspondence of states is maintained.

$\underline{a = send\text{-}seg_{cs}(SYN,\ sn_c)}$.

The corresponding execution fragment $\alpha = (u, \lambda, u')$ (recall that λ is the empty action sequence). The action a does not affect any variables involved in the mapping, so the correspondence of states is maintained.

$\underline{a = receive\text{-}seg_{cs}(SYN,\ sn_c)}$.

This effects of this step can be divided into two cases.

1. The first case occurs if in $\mathcal{TCP}^h$ $s.mode_s = \texttt{listen}$. For this case the corresponding $\alpha = (u,\ choose\text{-}server\text{-}id(j),\ u')$. In $\mathcal{TCP}^h$ if $s.mode_s = \texttt{listen}$, then $s'.mode_s = \texttt{syn-rcvd}$, $s'.ack_s = [sn_c] + 1$, $s'.sn_s = s.sn_s + 1$, $s'.isn_s = s'.sn_s$, $s'.used\text{-}id_s = s.used\text{-}id_s \cup \{s'.isn_c\}$. After the corresponding execution fragment α in D^p, $u'.id_s =$

$j \in SID$, $u'.used\text{-}id_c = u.used\text{-}id_c \cup \{u'.isn_c\}$, and $u.q\text{-}stat_{sc}(u'.id_s) = \texttt{live}$. These states are the correct ones as defined by the mapping R_{TD}. If $s.mode_s = \texttt{listen}$ then in the corresponding state u, $u.choose\text{-}sid = \texttt{true}$, so the *choose-server-id(j)* action is enabled.

2. The second case occurs for all other values of $s.mode_s$. For this case the corresponding $\alpha = (u, \lambda, u')$. If $s.mode_s = \texttt{closed}$, then $s'.send\text{-}rst_s$ gets set to $\texttt{true}$ and $s'.rst\text{-}seq_s$ gets set to 0, and $s'.ack_s$ is set to $[sn_c] + 1$. However, none of these assignments affect the mapping, so $u = u'$. For all other values of $s.mode_s$ a has no effect.

$\underline{a = send\text{-}seg_{sc}(SYN,\ sn_c,\ ack_s)}$.

The only effect of this step is to add the segment $(SYN,\ sn_c,\ ack_s)$ to $s.in\text{-}transit_{sc}$ which does not affect the mapping, so the corresponding execution fragment is $\alpha = (u, \lambda, u')$.

$\underline{a = receive\text{-}seg_{sc}(SYN,\ sn_c,\ ack_s)}$.

For this step the the corresponding $\alpha = (u, \lambda, u')$. Let the received segment be p. If $s.mode_c = \texttt{syn-sent}$ and $ack(p) = sn_c + 1$, this step changes $mode_c$ to $\texttt{estb}$, and ack_c to $sn(p) + 1$. It also adds $(isn_c, sn(p))$ to *estb-pairs*. These changes affect the mapping to $queue_{cs}(i)$, where $i = isn_c$, because after the these changes, we may have case (C) of the mapping, where in state s, we had case (B). Therefore, to show that the mapping is preserved after α and a, we need to show that $u.queue_{cs}(i) = u'.queue_{cs}(i)$. To do this we need to show that $s'.rcv\text{-}buf_s$ and $s'.cur\text{-}msg_c$ are both empty. If we do have case (C) of the mapping to $queue_{cs}(i)$ then by Invariant 7.19 we know $s'.mode_s = \texttt{syn-rcvd}$. If $s'.mode_s = \texttt{syn-rcvd}$, then Invariant 7.14 tells us that $s'.rcv\text{-}buf_s = \epsilon$. From Invariant 7.6 we know $s'.isn_c^s < s'.ack_s$, and from Invariant 7.10 we know that $s.isn_c = s.sn_c = s.isn_c^s$. Since this step does not change isn_c, sn_c, or isn_c^s, we know that after this step $s'.sn_c < s'.ack_s$. Therefore, $s'.cur\text{-}msg_c$ is empty. Thus, $u.queue_{cs}(i) = u'.queue_{cs}(i)$.

$\underline{a = prepare\text{-}msg_c}$.

For this step of $\mathcal{TCP}^h$ the corresponding execution fragment , α, in D^p is (u, λ, u'). It is clear that both a and α have the empty trace. We now show that $R_{\text{TD}}(s') = u = u'$. The action a changes sn_c and may change $mode_c$, msg_c and $send\text{-}buf_c$. It may change $mode_c$ to $\texttt{fin-wait-1}$, or $\texttt{last-ack}$, but only if $s.rcvd\text{-}close_c = \texttt{true}$ which means $u.mode_c =$

$u'.mode_c$ = `inactive`. The changes to sn_c, msg_c, and $send\text{-}buf_c$, may change the mapping of $u.queue_{cs}(i)$, but only for case (C) because the changes do not affect the mapping for cases (A) and (D), and states where a is enabled clearly do not overlap with case (B) because the precondition on action a requires $s.mode_c \in \{\texttt{estb}, \texttt{close-wait}\}$.

Therefore, we need to show that $u'.queue_{cs}(i) = u.queue_{cs}(i)$ for case (C). If we are in case (C) for $u.queue_{cs}(i)$ then $(i, isn_s) \in s.estb\text{-}pairs \;\wedge\; s.isn_c^s = i \;\wedge\; s.mode_s \notin \{\texttt{rec}, \texttt{reset}\}$, and if a is enabled then $((s.mode_c \in \{\texttt{estb}, \texttt{close-wait}\}) \wedge (s.send\text{-}buf_c \neq \epsilon \vee s.rcvd\text{-}close_c) \wedge \neg s.ready\text{-}to\text{-}send_c)$. Therefore, by Invariant 7.30 we know $s.ack_s > s.sn_c$, but we also know by Invariant 7.2 that $s.ack_s \leq s.sn_c + 1$. Therefore, we know that $s.ack_s = s.sn_c + 1$, which means $s.cur\text{-}msg_c$ is empty. If $s.send\text{-}buf_c$ is not empty then after a, $s'.sn_c = s'.ack_s$ or $s'.sn_c = s'.ack_s + 1$, so $s'.cur\text{-}msg_c = (head(s.send\text{-}buf_c) \times \texttt{ok})$. However, $s'.send\text{-}buf_c = tail(s.send\text{-}buf_c)$. Thus, $u'.queue_{cs}(i) = u.queue_{cs}(i)$, so the mapping is preserved. If $s.send\text{-}buf_c$ is empty, $s'.cur\text{-}msg_c$ is also empty, so again we get $u'.queue_{cs}(i) = u.queue_{cs}(i)$.

$\underline{a = prepare\text{-}msg_s}$.

This step is symmetric to $a = prepare\text{-}msg_c$, and the correspondence between this step and the empty step can be shown in a symmetric manner.

$\underline{a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)}$.

This is another step where the corresponding $\alpha = (u, \lambda, u')$, since a does not change any states that affect the mapping.

$\underline{a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)}$.

Let p be the segment received in this action. The effects of this step can be broken down into three cases based on the value of $s.mode_s$.

1. Case one occurs if $s.mode_s$ = `syn-rcvd` and $ack(p) = s.sn_s + 1$, then the corresponding α of D^p is *(u, make-assoc(i,j), u')*, where i is $u.id_c$ that corresponds to $s.isn_c$ and j is $u.id_s$ that corresponds to $s.isn_s$. Both a and α have the empty trace. The action *make-assoc(i,j)* is enabled in D^p because $s.isn_c \in s.ucid \;\wedge\; s.isn_s \in s.used\text{-}id_s$, so correspondingly in D^p $u.id_c \in u.ucid \;\wedge\; u.id_s \in u.used\text{-}id_s$, and by Invariant 7.33 neither $s.isn_c$ nor $s.isn_s$ are part of any pair already in $s.assoc$, so in the corresponding

state neither $u.id_c$ nor $u.id_s$ are part of any pair already in $u.assoc$. After a and α we have the correct correspondence between $s'.assoc$ and $u'.assoc$. The other variables that could get changed here are ack_s and $rcv\text{-}buf_s$ if $sn(p) = s.ack_s$. The changes being $s'.ack_s = s.ack_s + 1$ and $s'.rcv\text{-}buf_s = s.rcv\text{-}buf_s \cdot msg_c$. These changes affect the mapping of $u.queue_{cs}(i)$ for cases (C) and (D).

If we are in case (C), then Invariant 7.53 tells us that $sn_c = sn(p)$. Therefore, the change of ack_s means $s.cur\text{-}msg_c$ is $(s.msg_c, \texttt{ok})$ and $s'.cur\text{-}msg_c$ is empty. Invariant 7.36 tells us that if $s.msg_c \neq \texttt{null}$ and $sn(p) = s.sn_c$, then $msg(p) = s.msg_c$. Therefore, since $s'.rcv\text{-}buf_s = s.rcv\text{-}buf_s \cdot msg(p)$, $u.queue_{cs}(i) = u'.queue_{cs}(i)$.

For case (D), Invariant 7.62 tells us that there are no other segments on the channel with sequence number greater than $sn(p)$. Therefore, the change in ack_s means $s.p\text{-}pair_c$ is $\{(msg(p), sn(p))\}$ and $s'.p\text{-}pair_c$ is the empty set. However, as for case (C), since $s'.rcv\text{-}buf_s = s.rcv\text{-}buf_s \cdot msg(p)$ and Invariant 7.37 tells us that any segment with sequence number $sn(p)$ has the same message or the message is `null`. However, Invariants 7.55 and 7.56 tell us that any segment with sequence number $sn(p)$ has a message that is not `null`, so $u.queue_{cs}(i) = u'.queue_{cs}(i)$.

2. Case two occurs if $s.mode_s = \texttt{last-ack}$ and $ack(p) = sn_s + 1$. For this case α is $(u,\ set\text{-}nil_s,\ u')$. Clearly a and α both have the empty trace. We must show that $set\text{-}nil_s$ is enabled in state u of D^p. Since $s.mode_s = \texttt{last-ack}$, from our mapping we know $u.id_s \neq \texttt{nil}$ and from Invariant 7.13 we know that $u.mode_s = \texttt{inactive}$. The third part of the precondition requires that $\exists\ i$ s.t. $(i, u.id_s) \in\ u.assoc$. From Invariant 7.38 we know that since $s.mode_s = \texttt{last-ack}$, and there exists i such that $(i, s.isn_s) \in\ s.assoc$, so that part of the precondition holds for the corresponding state u.

   The fourth part of the precondition requires $u.queue_{cs}(i)$ to be empty. We only need to show this for cases (C) and (D) of the mapping to $u.queue_{cs}(i)$ because we know that there exists i such that $(i, s.isn_s) \in\ s.assoc$, which rules out queues for the other two cases.

We first examine case (C). Recall that the states for this case are states where $s.isn_c = i \ \wedge \ s.mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge \ (i, isn_s) \in s.estb\text{-}pairs \ \wedge s.isn^s_c = i \ \wedge \ s.mode_s \notin \{\texttt{rec}, \texttt{reset}\}$. To show that this queue is empty, we need to show that $s.send\text{-}buf_c$, $s.cur\text{-}msg_c$, and $s.rcv\text{-}buf_s$ are all empty. If $s.mode_s = \texttt{last-ack}$, and $u.queue_{cs}(i)$ is defined for case (C) then Invariant 7.52 tells us that $s.mode_c \in \{\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$, which coupled with Invariant 7.13 means $s.send\text{-}buf_c$ is empty. From Invariant 7.57 we know that that $s.sn_c < s.ack_s$, which means $s.cur\text{-}msg_c$ is empty. Finally, Invariant 7.59 indicates that $s.rcv\text{-}buf_s$ is empty. Therefore, $u.queue_{cs}(i)$ is empty.

Case (D) of the mapping to $u.queue_{cs}(i)$ occurs when $(s.isn_c \neq i \vee s.mode_c \in \{\texttt{rec}, \texttt{reset}\}) \wedge ((i, isn_s) \in s.estb\text{-}pairs \wedge s.isn^s_c = i \wedge s.mode_s \notin \{\texttt{rec}, \texttt{reset}\})$. To show that this queue is empty, we need to show that $s.p\text{-}pair_c$ and $s.rcv\text{-}buf_s$ are empty. From Invariant 7.65 we that for all non-SYN segments $q \in s.in\text{-}transit_{cs}$, $sn(q) < s.ack_s$ which means $s.p\text{-}pair_c$ is empty, and from Invariant 7.67 we know that $s.rcv\text{-}buf_s$ is also empty.

The fifth and final part of the precondition for the $set\text{-}nil_s$ action in D^p states that $(u.mode_c = \texttt{inactive} \ \vee \ u.id_c \neq \ i)$. From Invariant 7.70 we know this condition is true in state u.

After a, $s'.mode_s = \texttt{closed}$, and after α, $u'.id_c = \texttt{nil}$. Therefore, the mapping is preserved for this variable. The changes caused by a, and by α do not affect the mapping to $q\text{-}stat_{sc}(i)$. Since, we know there exists i such that $(i, s.isn_s) \in s.assoc$, and that $s.mode_s = \texttt{last-ack}$, only queues for case (C) of the mapping to $u.queue_{sc}(j)$, may exist in state s. However, since for this case, $s'.mode_s = \texttt{closed}$, $s'.isn_s = \texttt{nil}$, and $s'.ack_s$, $s'.msg_s$ and $s'.send\text{-}buf_s$ are all undefined, $u.queue_{sc}(s.isn_s)$ is affected by these changes. These chages take s' into the set of states for case (D) of the mapping for $queue_{sc}(j)$. However, since α does not change $u.queue_{sc}(s.id_s)$, we need to show that $u.queue_{sc}(s.id_s) = u'.queue_{sc}(s.id_s)$. In order to show this we need to show that $s.curmsg_s$ and $s.sbuf_s$ are empty and that $s'.p\text{-}pair_s$ is the emptyset. From Invariant 7.13 we know $s.send\text{-}buf_s = \epsilon$. Also since $[ack_c] = s.sn_s + 1$ and from Invariant 7.23 we know that $ack_c \geq [ack_c]$, so $s.cur\text{-}msg_s = \epsilon$. Since $[ack_c] = s.sn_s + 1$

and from Invariant 7.23 we know that $ack_c \geq [ack_c]$ and from Invariant 7.1 we know that for all $p \in$ *in-transit*$_{sc}$ $sn_s \geq sn(p)$, we know $s'.p\text{-}pair_s$ is the empty set, so $u.queue_{sc}(s.id_s) = u'.queue_{sc}(s.id_s)$.

3. The third case is for all other states s. The corresponding $\alpha = (u, \lambda, u')$. For this case, ack_s and $rcv\text{-}buf_s$ may change as in case one, and $mode_s$ may change from `fin-wait-1` to `fin-wait-2`, or from `closing` to `timed-wait`. The proof that the mapping for $u.queue_{cs}(i)$ is preserved is the same as case 1, and the possible changes to $mode_s$ in $\mathcal{TCP}^h$ do not affect its mapping to $mode_s$ in D^p.

<u>a = *send-seg*$_{sc}$(sn_s, ack_s, msg_s)</u>.

This is symmetric to a = *send-seg*$_{cs}$(sn_c, ack_c, msg_c).

<u>a = *receive-seg*$_{sc}$(sn_s, ack_s, msg_s)</u>.

This step is not quite symmetric to the step with a = *receive-seg*$_{cs}$(sn_c, ack_c, msg_c). This step has two case instead of three. However, the two cases are is basically symmetric to cases two and three of the symmetric step.

1. Case one occurs if $s.mode_c$ = `last-ack` and $[ack_s] = sn_c + 1$. For this case α is (u, *set-nil*$_c$, u'). Clearly a and α both have the empty trace. The proof that *set-nil*$_c$ is enabled in state u uses the same invariants as the proof that *set-nil*$_s$ is enable for the second case of the step with a = *receive-seg*$_{cs}$(sn_c, ack_c, msg_c) except that Invariant 7.46 is needed to show that there exists a j such that $(isn_c, j) \in assoc$.

   To show that the mapping is preserved for case (C) of $u.queue_{cs}(s.isn_c)$ after this step is also not quite symmetric to the proof that the mapping is preserved for case (C) of $u.queue_{sc}(s.isn_s)$ shown above for the symmetric step. We still have the queue going from case (C) to (D), and the proof that $s.send\text{-}buf_c$ is empty is symmetric. However, to show that $s.cur\text{-}msg_c$ is empty and $s'.p\text{-}pair_c$ is the empty set is not quite symmetric. To show both we use the fact that from Invariant 7.1 we know that for all $p \in$ *in-transit*$_{cs}$ $sn_c \geq sn(p)$, and from Invariant 7.24 we know that in the set of states were we have case (C) of the mapping to $queue_{cs}(i)$, $s.ack_s \geq [ack_s]$. Therefore, since $[ack_s] = sn_c + 1$, we know that $s.sn_c < s.ack_s$, and for all segments $p \in s'.in\text{-}transit_{cs}$, $s'.ack_s > sn(p)$. Therefore, $s.cur\text{-}msg_c = \epsilon$ and $s'.p\text{-}pair_c$ is the empty set.

2. Case two is for all other states s. The corresponding $\alpha = (u, \lambda, u')$. For this case the step may changed ack_c and $rcv\text{-}buf_c$ if $sn(p) = s.ack_c$. The changes being $s'.ack_c = s.ack_c + 1$ and $s'.rcv\text{-}buf_c = s.rcv\text{-}buf_c \cdot msg_c$. These changes affect the mapping of $u.queue_{sc}(j)$ for cases (C) and (D).

   If we are in case (C), then Invariant 7.53 tells us that $sn_s = sn(p)$. Therefore, the change of ack_c means $s.cur\text{-}msg_s$ is $(s.msg_s, \texttt{ok})$ and $s'.cur\text{-}msg_s$ is empty. Invariant 7.36 tells us that if $s.msg_s \neq \texttt{null}$ and $sn(p) = s.sn_s$, then $msg(p) = s.msg_s$. Therefore, since $s'.rcv\text{-}buf_c = s.rcv\text{-}buf_c \cdot msg(p)$, $u.queue_{sc}(j) = u'.queue_{sc}(j)$.

   For case (D), Invariant 7.62 tells us that there are no other segments on the channel with sequence number greater than $sn(p)$. Therefore, the change in ack_s means $s.p\text{-}pair_s$ is $\{(msg(p), sn(p))\}$ and $s'.p\text{-}pair_s$ is the empty set. However, as for case (C), since $s'.rcv\text{-}buf_c = s.rcv\text{-}buf_c \cdot msg(p)$ and Invariant 7.37 tells us that any segment with sequence number $sn(p)$ has the same message or the message is $\texttt{null}$. However, Invariants 7.55 and 7.56 tell us that any segment with sequence number $sn(p)$ has a message that is not $\texttt{null}$, so $u.queue_{sc}(j) = u'.queue_{sc}(j)$.

<u>$a = receive\text{-}msg_c(m)$.</u>

For this step the corresponding $\alpha = (u, a, u')$. We first need to show that $receive\text{-}msg_c(m)$ is enabled in state u. This step only affects the mapping of $u.queue_{sc}(j)$, for cases (C) or (D). Since we have $s.mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge head(s.rcv\text{-}buf_c) = m \ \wedge \ m \neq \texttt{null}$, it is clear that in the corresponding state we have $\neg u.rec_c \wedge head(u.queue_{sc}(j)) = m \ \wedge \ m \neq \texttt{null}$. For both cases (C) and (D) $(s.isn_c, j) \in s.assoc$. Therefore, in the corresponding state $(u.id_c, j) \in u.assoc$, and $u.q\text{-}stat_{sc}(j)$ is $\texttt{live}$. Therefore, this action is enabled in state u. It is easy to see that the mapping to $queue_{sc}(j)$ is preserved after this step.

<u>$a = receive\text{-}msg_s(m)$.</u>

For this step the corresponding $\alpha = (u, a, u')$. Showing that we can simulate this step in D^p is not quite symmetric to the previous case because the conditions for cases (C) and (D) for the mapping to $queue_{cs}(i)$ is not symmetric to the same cases for the mapping to $queue_{sc}(j)$. Since we have $s.mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge head(s.rcv\text{-}buf_s) = m \ \wedge \ m \neq \texttt{null}$, it is clear that in the corresponding state we have $\neg u.rec_s \wedge head(u.queue_{cs}(i)) = m \wedge m \neq \texttt{null}$. Since for these cases we know $mode_s \notin \{\texttt{closed}, \texttt{rec}, \texttt{reset}\}$, and we know from Invariant 7.14 that if $mode_s \in \{\texttt{listen}, \texttt{syn-rcvd}\}$ that $rcv\text{-}buf_s$ is empty, we know $s.mode_s \in sync\text{-}states$. For both cases Invariants 7.38 and 7.44 tells us that $(i, isn_s) \in assoc$. Therefore, in the corresponding state $(i, u.id_s) \in u.assoc$, and $u.q\text{-}stat_{cs}(i)$ is `live`. Therefore, this action is enabled in state u. It is easy to see that the mapping to $queue_{cs}(i)$ is preserved after this step.

$a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$.

The effects of this step do not affect the mapping, so the corresponding α is (u, λ, u').

$a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$.

This step affects the mapping in a manner similar to the step with $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$. However, since none of the possible effects of this step causes the server to close, we only have two cases. Let p be the segment received in this step.

1. This case is if $s.mode_s = \texttt{syn-rcvd}$ and $sn(p) \geq s.ack_s$ and $ack(p) = s.sn_s + 1$. The corresponding $\alpha = (u,\ make\text{-}assoc(i,j),\ u')$. The proof of correspondence is the same as for first case of $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$

2. Case two is for all the states. For this case $\alpha = (u, \lambda, u')$. The proof of correspondence is the same as the proof for case three of $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$.

$a = send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$.

This step does not affect the mapping, so the corresponding α is (u, λ, u').

$a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$

For this case $\alpha = (u, \lambda, u')$. This step affects the mapping is the same manner as the second case for the step with $a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$. The proof that the mapping is preserve by after α is the same as the proof for that case.

$a = timeout_c$.

The corresponding α is $(u,\ set\text{-}nil_c, u')$, and both steps have the empty trace. The resulting states also clearly correspond. The difficulty in showing the correspondence, as it was for case for $a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$, is in showing that $set\text{-}nil_c$ is enabled in state u. The same conditions that were true for that case holds for this case except one. We need a different invariant to show that $s.rcv\text{-}buf_c$ is empty. We know that $s.rcv\text{-}buf_c$ is empty because of Invariant 7.60.

$\underline{a = timeout_s}$.

This step is symmetric to $a = timeout_c$.

$\underline{a = crash_c}$.

The corresponding α in the delayed specification automaton D^p is the following sequence of steps $(u,\ crash_c,\ u''',\ mark_c(I),\ u'',\ drop_c(I', k),\ u')$. Clearly, α has the same trace as a since $crash_c$ is the only external action in the sequence.

First we show that this sequence of steps is enabled in D^p. After $crash_c$, rec_c is `true`, so $mark_c(I)$ is enabled, and $drop_c(I', k)$ is enabled if I' and k are defined correctly. We define I, I', and k below and show that $\mathrm{R}_{TD}(s')$ is indeed the state u' we get after the sequence of steps α.

The only change in state caused by a is that $s'.mode_c$ = `rec`. This change affects the mapping of $u.mode_c$, $u.q\text{-}stat_{sc}(j)$, $u.queue_{sc}(j)$, and $u.queue_{cs}(i)$. It is easy to see that the mapping of $mode_c$ is preserved. We have $s'.mode_c$ = `rec` and $u'.rec_c$ = `true`, which is correct by R_{TD}. For $u.q\text{-}stat_{sc}(j)$ and $u.queue_{sc}(j)$, α preserves the mapping because if $(s.isn_c, j) \in assoc$ then $u'.q\text{-}stat_{sc}(j)$ is `dead` and $u'.queue_{sc}(j)$ is empty, which is correct by R_{TD}. Otherwise, α does not change $u.q\text{-}stat_{sc}(j)$ or $u.queue_{sc}(j)$, and a does not affect the mapping. To show that the mapping of $u.queue_{cs}(i)$ is preserved is more complicated. We break the possible states into two cases. We define I and I' for each case. Note that the $mark_c(I)$ and $drop_c(I', k)$ actions do not affect the mapping of $u.mode_c$, $u.q\text{-}stat_{sc}(j)$, or $u.queue_{sc}(j)$, so these mappings are not affected by the different values of I, I', and k.

1. The first case is for cases (A),(B) (D) of the mapping to $queue_{cs}(i)$. For these cases $I = I' = \emptyset$ and $k = i$, so α does not change $u.queue_{cs}(i)$. The correspondence of states is preserved because a does not affect the mapping for these queues. It is obvious that for case (A) of the mapping to $queue_{cs}(i)$ a has no effect. This is also the case for case (B) because a only changes $s.mode_c$ to `rec` which is one of the modes in which $u.queue_{cs}(i)$ exists. For case (D) of the mapping to $queue_{cs}(i)$, the fact that a changes $s.mode_c$ to `rec` could change the queue because it could cause $s.p\text{-}pair_c$ to go from the

empty set to having an element. This change only happens for in this case if $s.isn_c \neq i$, since the other set of conditions for case (D) requires that $s.mode_c$ already be in the set $\{\texttt{rec}, \texttt{reset}\}$. However, if $s.isn_c \neq i \wedge (i, s.isn_s) \in s.estb\text{-}pairs \wedge s.isn_c^s = i$ then Invariant 7.34 tells us that $s.mode_c \in \{\texttt{rec}, \texttt{reset}, \texttt{closed}, \texttt{syn-sent}\}$, so assigning $s'.mode_c$ to `rec` does not affect the mapping for this case.

2. We now examine case (C) of the mapping to $queue_{cs}(i)$. If the we are in case (C) of the mapping in state s, then after action a we go to case (D) of the mapping to $queue_{cs}(i)$. We can break this case into two subcases based on whether $s'.p\text{-}pair_c$ is empty or not. For both subcases $i = k$. We use the following preliminary definition: $suffix_{rb} = \{i \| \text{s}.rcv\text{-}buf_s| < i \leq maxindex(u.queue_{cs}(i))\}$. That is, $suffix_{rb}$ is the suffix of $u.queue_{cs}(i)$ that starts with the element that maps to the first element after $s.rcv\text{-}buf_s$.

   (a) If there exists a segment $p \in s.in\text{-}transit_{cs}$, where p is of type $(sn_c,\ ack_c,\ msg_c)$ and $sn(p) = s.ack_s$ or there exists a segment $q \in s.in\text{-}transit_{cs}$, where q is of type $(sn_c,\ ack_c,\ msg_c,\ FIN)$ and $sn(q) = s.ack_s + 1$, then $s'.p\text{-}pair_c \neq \emptyset$. Therefore, $I = suffix_{rb}$ and $I' = suffix_{rb}/maxindex(suffix_{rb})$. I' is the suffix of $u.queue_{cs}(i)$ that starts with the element that maps to the second element after $s.rcv\text{-}buf_s$ which is also the first element after $s'.p\text{-}pair_c$. After a, we have case (D) of the mapping to $queue_{cs}(i)$, but since α deletes all the elements after $s'.p\text{-}pair_c$, we get the right corresponding state.

   (b) Case two occurs for all other states for case (C). That is, states where $s'.p\text{-}pair_c = \emptyset$. For this case $I = I' = suffix_{rb}$. After α $u'.queue_{cs}(i)$ corresponds to the $s'.rcv\text{-}buf_s$. However, this still satisfies the mapping of $u'.queue_{cs}(i)$ for case (D) because $s'.p\text{-}pair_c$ is empty.

$\underline{a = crash_s}$.

This step is symmetric to $a = crash_c$, except in the arguments required to show that case (D) of the mapping to $queue_{sc}(j)$ is preserve. For this case we need to show the symmetric thing. That is, changing $mode_s$ to `rec` does not affect the mapping for this case. Therefore, we need to show that if $s.isn_s \neq j \ \wedge \ (s.isn_c, j) \in assoc$ then $s.mode_s \in \{\texttt{closed}, \texttt{listen}, \texttt{syn-rcvd}, \texttt{rec}, \texttt{reset}\}$. From Invariants 7.38 and 7.44 we know that if $mode_s \in$ *sync-states* then $(isn_c, isn_s) \in assoc$, and from Invariant 7.33 we know isn_c is only paired with on value of j. Therefore, since for this case we know $isn_s \neq j$, we know $s.mode_s \notin$ *sync-states*. Thus, the mapping is preserved for this case.

<u>$a = recover_c$.</u>

The corresponding α of D^p is *(u, mark$_c$(I), u''', drop$_c$(I,k), u'', recover$_c$, u')*. Since only $recover_c$ is external, the traces of a and α are clearly the same. We first show that this sequence of steps is enabled in D^p. The action $recover_c$ is enabled in $\mathcal{TCP}^h$ if $s.mode_c = \texttt{rec}$. This state maps to $u.rec = \texttt{true}$ in which case *mark$_c$(I)* is enabled and *drop$_c$(I,k)* is also enabled. Since neither *mark$_c$(I)* nor *drop$_c$(I,k)* changes $u.rec$, then $recover_c$ is also enabled. We define I and k appropriately below.

We now show that the state of D^p we get after α is the same as $R_{\mathrm{TD}}(s')$. After a, $s'.mode_c = \texttt{closed}$. This change affects the mapping for $u.rec_c$, $u.id_c$, $u.queue_{cs}(i)$, and $u.q\text{-}stat_{cs}(i)$. After α $u'.rec_c = \texttt{false}$ and $u'.id_c = \texttt{nil}$, so the mapping is preserved for those variables. For $u.queue_{cs}(i)$ the mapping is only affected by a if state s is in case (B) of the mapping to $queue_{cs}(i)$, because case (C) does not hold if $s.mode_c = \texttt{rec}$, and for cases (A) and (D) the action does not affect the mapping. It does not affect the mapping for case (A), because the change of $mode_c$ from rec to closed does not affect the conditions for this case. It does not affect the the mapping for case (D) because the change can only cause the condition for this case to go from $s.mode_c = \texttt{rec}$ to $s.isn_c \neq i$, but this does not change the fact that it is case (D), nor does it change the contents of the queues. Therefore, $u.queue_{cs}(i)$ for cases (A) and (D), $I = \emptyset$ and $k = i$.

For case (B) of the mapping to $u.queue_{cs}(i)$, after action a it is in group (A). Let $I = dom(u.queue_{cs}(i))$ and $k = i$. The mapping is preserved because after a, $s.send\text{-}buf_c$ is deleted and after α all the elements are deleted. Finally, to show that the mapping for $q\text{-}stat_{cs}(i)$ is preserved we note that if $i = s.isn_c \ \wedge \ (i, s.isn_c) \notin \ s.assoc$, then after a, $u'.q\text{-}stat_{cs}(i)$ maps to dead. For this state s, the corresponding state u satisfies $\forall\ j(u.id_c, j) \notin \ u.assoc$, and in $u''.queue_{cs}(i) = \epsilon$. Therefore, after α $u'.q\text{-}stat_{cs}(i) = \texttt{dead}$.

$\underline{a = recover_s}$.

This step is symmetric to $a = recover_c$.

$\underline{a = drop_{cs}(p)}$ (from the $Ch_{cs}(\mathcal{P})$ component of $\mathcal{TCP}^h$).

There are two case for this step.

1. The first case occurs if p is the only copy of itself in $s.in\text{-}transit_{cs}$, $s.mode_c \in \{$`rec`, `reset`, `closed`, `syn-sent`$\}$, and if p is not a FIN segment $sn(p) = s.ack_s$ and if it is a FIN segment $sn(p) = s.ack_s + 1$. That is, if $\{(sn(p), msg(p))\} = s.p\text{-}pair_c$, and it is the last copy of this segment on $s.in\text{-}transit_{cs}$. For this case the corresponding α is $(u,\ drop_c(I,k), u')$. Since both actions are internal, the traces are the same. This is a case where a message that exists during $crash_c$ may get lost after $recover_c$ which is why we used the Delayed Decision Specification. From our definition $s.p\text{-}pair_c$ only exists if $s.mode_c \in \{$`rec`, `closed`, `syn-sent`$\}$. The action a only affects the mapping of $u.queue_{cs}(i)$ for case (D). Let $I = maxindex(u.queue_{cs}(i))$ and $k = i$. This is the last element of the queue and it is also the element that corresponds to $(message(s.p\text{-}pair_c) \times$ `marked`$)$. Since this element is a suffix of $u.queue_{cs}(i)$ and it is `marked`, $drop_c(I,k)$ is enabled and clearly produces the right corresponding state.

2. The second case is for all other states. For these states the corresponding $\alpha = (u, \lambda, u')$. Clearly the traces are the same, and for these states of $\mathcal{TCP}^h$, a does not affect the mapping, so the resulting states correspond.

$\underline{a = drop_{sc}(p)}$ (from the $Ch_{sc}(\mathcal{P})$ component of $\mathcal{TCP}^h$) .

This step is symmetric to $a = drop_{cs}(p)$.

$\underline{a = duplicate_{cs}(p)}$ (from the $Ch_{cs}(\mathcal{P})$ component of $\mathcal{TCP}^h$).

The corresponding α is (u, λ, u'). Since a is internal we have the same trace. The only aspect of the mapping one might think a would affect is $s.p\text{-}pair_c$. However, since $s.p\text{-}pair_c$ is a set, duplication has no effect.

$\underline{a = duplicate_{sc}(p)}$ (from the $Ch_{sc}(\mathcal{P})$ component of $\mathcal{TCP}^h$).

This step is symmetric to $a = duplicate_{cs}(p)$.

$a = \nu(t)$ *(time-passage)*

The corresponding α in D^p is $(u, \nu(t), u')$, the time-passage action of the patient Delayed Decision Specification.

$a =$ *send-seg*$_{sc}$*(RST, ack*$_s$*, rst-seq*$_s$*)*.

For this step the corresponding step α of D^p is (u, λ, u'). Clearly the traces are the same, since *send-seg*$_{sc}$*(RST, ack*$_s$*, rst-seq*$_s$*)* is an internal action. The only change made by a to the state is that s'.*send-rst*$_s$ is `false`. This variable does not affect R_{TD}, so $u = u'$.

$a =$ *receive-seg*$_{sc}$*(RST, ack*$_s$*, rst-seq*$_s$*)*.

For this step we have two cases based on the value of $s.mode_c$. The two cases are as follows.

1. The first case occurs if $s.mode_c =$ `closed` or $s.rst\text{-}seq_s \neq s.ack_c$ or if $s.rst\text{-}seq_s = 0$ and $s.ack_s \neq s.sn_c + 1$. For this case $\alpha = (u, \lambda, u')$. This is correct because in this state, a has no effect.

2. Case two is for all other states. For this case the corresponding α in D^p is the following sequence of steps *(u, abort*$_c$*, u‴, mark*$_c$*(I), u″, drop*$_c$$(I', k)$*, u′)*. Clearly, α has the same trace as a, since all the actions of α are internal and a is internal. Since $abort_c$ is enabled if $u.id_c \neq$ `nil` it will be enabled from state u, and since it sets $abrt_c$, $mark_c(I)$ is enabled in u'''. After $mark_c(I)$, $drop_c(I', k)$ is enabled for the appropriately defined I' and k in state u''. We define I, I', and k below and show that mapping is preserved.

   The change caused by a is to make $s'.mode_c =$ `reset`. Apart from the fact that this change affects the mapping to $abrt_c$ and not rec_c, other effects on the mapping of this change is exactly the same as the changes caused by the $crash_c$ action. Furthermore, $abort_c$ has the exact same effect has $crash_c$ in specification D. Therefore, we can define I, I', and k, as they are define for the case of $a = crash_c$, and the proof that the mapping is preserved after α is also the same as for that case.

$a =$ *send-seg*$_{cs}$*(RST, ack*$_c$*, rst-seq*$_c$*)*.

This step is symmetric to $a =$ *send-seg*$_{sc}$*(RST, ack*$_s$*, rst-seq*$_s$*)*.

$a =$ *receive-seg*$_{cs}$*(RST, ack*$_c$*, rst-seq*$_c$*)*.

This step is not exactly symmetric to $a =$ *receive-seg*$_{sc}$*(RST, ack*$_s$*, rst-seq*$_s$*)*, but is quite similar. For this step we have two cases based on the value of $s.mode_s$.

1. Case one occurs if $s.mode_s \in \{\texttt{closed}, \texttt{rec}\}$ or $s.rst\text{-}seq_c \neq s.ack_s$. For this case α is (u, λ, u'). This simulation is clearly correct since for this state action a has no effect.

2. Case two is for all other reachable states. This case is symmetric to case 2 for $a =$ *receive-seg*$_{sc}$*(RST,* ack_s, $rst\text{-}seq_s$*)*.

$\underline{a = shut\text{-}down_c}$.

The corresponding α of D^p is *(u,* $mark_c(I)$*,* u'''*,* $drop_c(I,k)$*,* u''*,* $shutdown_c$*,* u'*)*. Since only *shut-down*$_c$ is internal, and all the actions in α are also internal the traces of a and α are clearly both the empty trace. We first show that this sequence of steps is enabled in D^p. The action *shut-down*$_c$ is enabled in $\mathcal{TCP}^h$ if $s.mode_c = \texttt{reset}$. This state maps to $u.abrt = \texttt{true}$ in which case $mark_c(I)$ is enabled and $drop_c(I,k)$ is also enabled. Since neither $mark_c(I)$ nor $drop_c(I,k)$ changes $abrt_c$, then *shut-down*$_c$ is enabled in state u''. We now need to define I and k, and show that the $R_{\rm TD}$ is preserved in state u'.

The effect of *shut-down*$_c$ in $\mathcal{TCP}^h$ is exactly the same as the effect of $recover_c$ in $\mathcal{TCP}^h$, and the effect of *shut-down*$_c$ in D^p is exactly the same as the effect of $recover_c$ in D^p except for the fact that *shut-down*$_c$ sets $abrt_c$ to `false` and $recover_c$ sets rec_c to `false`. Therefore, we can define I and k exactly as they are define for the above case of $a = recover_c$, and the proof that the mapping is preserved remains the same.

$\underline{a = shut\text{-}down_s}$.

This case is symmetric to the case for $a = shut\text{-}down_c$.

This concludes the simulation proof. ∎

7.3.3 Proof of trace inclusion

We can now proof that the GTA model of TCP, $\mathcal{TCP}$, implements a patient version of Specification S.

Theorem 7.1

$\mathcal{TCP} \sqsubseteq_t patient(S)$.

Proof: From Lemma 7.1 we get that $\mathcal{TCP}^h \leq_R^t D^p$, which because of the soundness of timed refinement mapping (Theorem 3.6) and the soundness of adding history variables (Theorem 3.9) implies that $\mathcal{TCP} \sqsubseteq_t D^p$. From Theorem 4.1 we know $D \sqsubseteq S$. Using the *Embedding Theorem* of [31] presented in Chapter 3 we now get $D^p \sqsubseteq_t \mathit{patient}(S)$. Thus, we now have $\mathcal{TCP} \sqsubseteq_t D^p$ and $D^p \sqsubseteq_t \mathit{patient}(S)$. Therefore, since the subset relation and thus the implements relation is transitive we get $\mathcal{TCP} \sqsubseteq_t \mathit{patient}(S)$. ∎

Chapter 8

TCP with bounded counters

In the previous chapter we proved the correctness of TCP if the protocol uses unbounded and stable counters. These counters guarantee that whenever a new incarnation is started, the segments for the incarnation are numbered with sequence numbers that had never been used before. The uniqueness of these sequence numbers prevented confusion with segments from previous incarnations. Therefore, old duplicate segments are not accepted and current segments are not rejected.

However, in practice there is no infinite source of uid's, and TCP uses a bounded cyclic number space for sequence numbers. TCP uses a clock based counter for initial sequence number (ISN) selection, and the number space is sufficiently small and the rate of change of the clock sufficiently fast that cycling through the number space is not uncommon. Additionally, in TCP the same number space that is used for the ISN's is also used for sequence numbers for each segment. That is, after an ISN is chosen, each subsequent packet is numbered starting from that ISN. Here again it does not require extreme conditions for a TCP host to send enough segments to cause the number space to cycle. In addition to being bounded, the counter used for ISN selection and the local sequence number variable of a host may not be stable. Therefore, after a crash a host may not know the last sequence number used, and the clock based counter may get reset to some arbitrary number. Consequently, when a new sequence number is chosen for a TCP segment, one cannot guarantee that this number has never been used before. However, uniqueness of new sequence numbers, as we will show, is not necessary for the protocol to behave correctly. A weaker property is sufficient. The property that is needed is that when a sequence number is chosen for a segment, there should not be a different segment with the same sequence number on the outgoing channel, nor a segment on the incoming channel that acknowledges the new sequence number, nor should the other host be able to acknowledge the new sequence number before it receives a new segment with that number. Having an unbounded counter that is increased for every new segment obviously achieves this property, but TCP uses a combination of the three-way handshake protocol and several timing mechanisms to achieve the same property. The timing mechanisms are used to ensure that when a sequence number is reused, old segments that have the same number are dropped from channels. The property is what is referred to as the "id not-in-use" condition in [27].

8.1 Timing constraints

Along with the three-way handshake protocol, TCP relies on timing properties of the bounded counters and the channels, and various timeouts to guarantee the id not-in-use condition. We first present the set of properties related to the counter used for ISN selection. Initial sequence numbers are chosen using 32 bit clock based counters. The low order bit of the clocks are incremented roughly every four microseconds. Thus, it takes approximately 4.5 hours for the clocks to cycle. The rate of change of the clocks must be faster than the time it takes to start a new incarnation of a connection. That is, it must take more than four microseconds for a host to close and reopen. Thus, between the closing and reopening of a host the clocks must tick at least once. Once an ISN is chosen, it is also the starting point for numbering segments sent by the host for the incarnation. In order for the protocol to work correctly, there must be a bound on the maximum rate of data transmission. This bound gives a bound on how fast the generation of new sequence numbers can cause the number space to cycle. The bound that is needed here goes back to the id-not-in use property. TCP host should not be able to send data at a rate where a number may be reused while an old segment with that sequence number or an old acknowledgment of a segment with that sequence number might still be on the channel.

The next timing property TCP relies on for correctness is the maximum segment lifetime

(MSL). The maximum segment lifetime is maximum period a segment can stay on a channel before it is dropped or delivered. For TCP this period taken to be two minutes. In practise the maximum segment lifetime in the net is not likely to exceed a few tens of seconds [28], and the maximum segment lifetime of two minutes is not strictly enforced in the net. However, for correctness purposes we assume this maximum is enforced. We refer to the duration of the maximum segment lifetime as μ.

In addition to the properties of the counters and the MSL, the following timeout mechanisms to ensure correct behavior in TCP.

1. When a connection closes normally (without a crash, reset, or timeout), one or both hosts remain in `timed-wait` state for a period of 2μ. Therefore, a new incarnation of the connection cannot be formed before all the segments from the previous incarnation are dropped from the channels.

2. When a host sends a segment that requires an acknowledgment, it starts a timer, and if a period of wt passes and it does not receive an acknowledgment of that segment, it stops sending the segment for a period of μ. If it still does not get a response in that period it closes. We refer to wt as the *maximum wait time*.

3. When a hosts receives an input that needs a response, it can wait for a maximum period of rt, before sending a response. The client also has a maximum delay of rt, before it must send a SYN segment after it receives the active open input from the user[1]. We refer to rt as the *maximum response time*.

4. The cycle time, maximum wait time, maximum response time, and MSL have the following relationship: $ct > 2(wt + rt + \mu)$.

5. If data is sent at a maximum rate, then the time to cycle through the number space is greater than $wt + rt + 2\mu$.

6. When a host receives a reset segment, it stops performing any actions for a period of at least μ before it closes.

[1]Technically, there is no need for a maximum delay for a response to this input. However, there must be a maximum wait before the client times out if it does not get a response to a SYN segment after it gets this input. Having a maximum response to this input coupled with the wait timeout of wt gives a maximum wait after the active open input of $wt + rt$.

7. Finally, after it crashes, a hosts must observe a quiet time, qt, where $qt > 2\mu + rt + wt$, in which it does nothing before it is allowed to reopen.

To prove that under normal close situations at least one host remains in `timed-wait`, is very complicated. Therefore, we make the assumption that when a host closes from mode `last-ack` there must be a period greater than μ before it is allowed to reopen. This assumption means that whenever a host closes and reopens, there are not segments from the previous incaranation on the outgoing channel.

The informal TCP specification [28, 30] states the cycle time of the clock based counter, states the time it takes for a host to cycle through the sequence number should be greater than the maximum segment lifetime, and gives the maximum segment lifetime of two minutes. It also recommends that the *quiet time* after a crash be the MSL. However, in [27] Murphy and Shankar point out that this period of time is not sufficient to guarantee that old duplicates are not received after a crash, and they state that the duration of qt that we present above is needed. Because this duration of quiet time is necessary, it means that TCP hosts must also have rt and wt timeout values. However, these values are not clearly specified in the informal TCP specification.

In [28] it states that when a connection is opened, the user has the option to include a timeout for all data submitted to TCP. If the data is not submitted to the destination within the timeout period, the connection is aborted. If the user does not specify a timeout, a global default of five minutes is used. In [30] which specifies host requirements for TCP, it says there must be two thresholds $R1$ and $R2$ for handling excessive retransmission of data segments. These values can be either time or number of retransmission. When the first threshold is reached or exceeded, the IP layer is notified, and when the second threshold is reached or exceeded the connection is closed. Applications (external users), must be able to set the value for $R2$ for a particular connection. In [30] an interactive application is cited as an example where $R2$ might be set to "infinity," giving the user control over when to disconnect. Such a setting for R2 would mean that there is no correct setting for qt as we defined it. However, the maximum wait timeout we present is only for the acknowledgment of data. Therefore, in the interactive situation when data is received by a host, even if response data is required, the host can still immediately send an acknowledgment. Once the acknowledgment is received the timer is turned off at the sender, and the application can wait indefinitely for the response data.

Having timeouts for excessive retransmission means that the protocol does not have the liveness property that if there are no crashes then all data gets delivered. For example, a channel can drop all the segments it receives for the maximum wait period. In this situation, the data on that segment does not get delivered even though there are no crashes. However, in practice the probability that a channel drops all segments sent for a period of wt is very low unless there is a partition in the network.

As for a maximum response time, the closest mention of this in the informal specification is in [30] where the issue of delaying acknowledgments is discussed. Delaying acknowledgments increases efficiency in "real" TCP which uses the sliding window protocol where each segment does not have to get an acknowledgment. In [30] it states that the delay must be less that 0.5 seconds. This is essentially a maximum response delay.

In practice five minutes is a reasonable default value for wt since by [30], a value less than 0.5 seconds for rt is required and RTO is usually much smaller than MSL which is two minutes. However, one wants to allow for the possibility that the data segment and the response segment take the full two minutes of the MSL. On the other hand, it is important for wt to be as low as is reasonable since the quiet time after a crash should be minimized. A value of five minutes for wt also satisfies the property that $ct > 2(wt + rt + \mu)$.

The waiting period of μ that we require a TCP host to observe after a maximum wait timeout or after a reset is needed so that the host cannot reopen before all the segments it sent for the previous incarnation are gone from the channels. This wait serves essentially the same purpose has timed wait state after a normal close. In the informal specification [28, 30] such a wait is not mentioned.

8.2 Duplicate delivery without timeouts

In this section we present some scenarios where the timeouts mentioned in the previous section are needed to prevent duplicate delivery of data. There are basically two types of situations where there is the potential for duplicate delivery of data — crash situations and *long-lived* connections. A long-lived connection is one that remains open for a period greater than or equal to ct. For some applications such as telnet, long-lived connections are not uncommon. If a connection is open for a period less than the cycle time and there are no crashes, then because a host cannot send data faster than the rate at which the clock counter ticks, if the connection is reopened the new ISN will be bigger than any sequence number from the previous incarnation. However, when the clock counter has cycled during the life of the connection, the new ISN might be equal to or less than sequence numbers of segments from the previous incarnation that might still be in the channels. Therefore, segments from the old incarnation might get confused with segments from the new incarnation.

In this section we present four examples of executions that may cause incorrect behavior if the right timeouts are not in place. The first three examples involve long-lived connections, and they require the client to choose the same ISN for successive incarnations. While certainly possible, the probability of the client choosing the same ISN for successive incarnations is extremely low. Therefore, the examples we show here are not likely to happen in practice, even without the proper timeouts, but are certainly possible. The fourth example involves an execution where after a crash, if the duration of quiet time is equal to the MSL, then data from a previous incarnation is delivered in a current incarnation.

The first example is shown in Figure 8-1. In this execution the client and server both receive the signal to open and then for a period that is very close to ct neither side sends a packet. When the client opens it reads the value i from the clock counter for its new ISN. When almost ct time has elapsed since the client received the signal to open there is a burst of activity.

The client sends a SYN segment to the server with sequence number i. That is, it sends a segment of the type *(SYN, sn_c)* where $sn_c = i$. When the server receives this segment, it reads its clock counter and gets ISN, j. It then sends a SYN plus acknowledgment segment to the client, *(SYN, sn_s, ack_s)*, where $sn_s = j$ and $ack_s = i{+}1$. Next the client acknowledges the SYN segment from the server and sends some data $d1$ by sending a *(sn_c, ack_c, msg_c)* segment, where $sn_c = i + 1$, $ack_c = j + 1$, and $msg_c = d1$. The server responds with a FIN segment that has sequence number $j + 2$, and data $d2$. When the client receives the FIN segment it goes to mode `close-wait`, and when it receives the signal to close it goes to

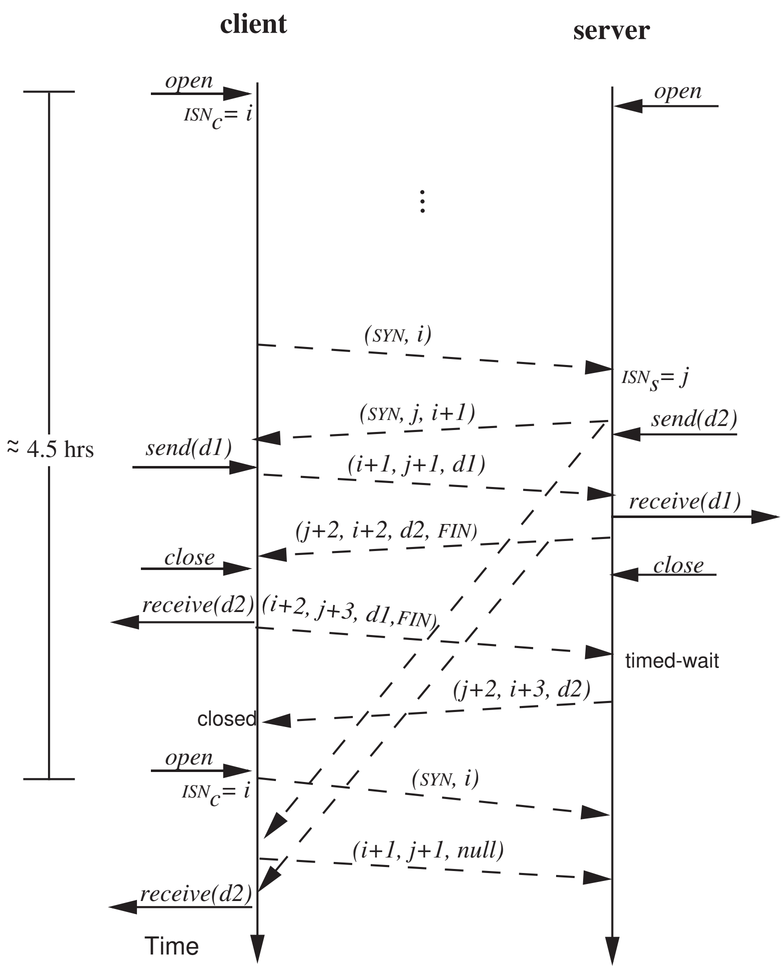

Figure 8-1: ***Example one.*** *An execution where a long delay after the open signals are received, causes the same message to be delivered twice.*

mode `last-ack`. The client also delivers data $d2$ and sends a FIN segment with sequence number $i+2$. When the server receives the FIN segment it goes to mode `timed-wait` and sends and acknowledgment segment. When the client receives this segment it can close immediately. Next the client gets a new signal to open, and because the clock counter has cycled since the last time the client chose an initial sequence number, it reads i again. The client can immediately send a SYN segment with ISN i. After it sends this segment, it receives a duplicate of the *(SYN, j, i+1)* segment the server sent earlier. It is possible that such a duplicate may still be on the channel because in this execution, a period of less than μ has elapse since the segment was first sent. When the client receives this segment, it accepts it as a valid acknowledgment of its SYN segment, and sends an acknowledgment to the server. After sending this segment the client receives a second duplicate segment from the previous incarnation. This segment is the FIN segment with data $d2$. When the client receives this segment, it is also accepted and the client may deliver data $d2$ again.

This duplicate delivery of data does not happen if the client has to send the SYN segment within rt of receiving the the *open* input, and times out if it does not get a response after a period of wt of sending the segment.

The second example is shown in Figure 8-2. In this execution, the client and server perform the three-way handshake immediately after they receive the signals to open. After the opening phase, neither side sends any segment for a period that is approximately equal to the cycle time ct. After this period of inactivity, the client gets an input to send message m from the user. It sends the data with a *(sn_c, ack_c, msg_c)* segment, where $sn_c = i+1$, $ack_c = j+1$, and $msg_c = m$. When the server receives this segment, it sends an acknowledgment that gets dropped from the channel. The server also passes m to the user. After the data is passed to the user, the server crashes, and after the recommended quiet time of μ the server recovers and goes to mode `closed`. Meanwhile, the client repeatedly retransmits the *(sn_c, ack_c, msg_c)* segment because it has not received an acknowledgment from the server, and each retransmission is dropped from the channel. However, after the server recovers, one of the retransmitted segments does not get dropped and arrives at the closed server. Because, the server is closed when the segment arrives, a reset is generated. Before, the reset reaches the client, it sends another copy of the *(sn_c, ack_c, msg_c)* segment which gets delayed on the channel. After sending the reset, the server re-opens. The client closes when it receives the reset, but immediately gets the open input from the user. The new open input from the user comes just as the clock counter cycles, so the client reads $isn_c = i$ again. Thus, the client again sends *(SYN, sn_c)* where $sn_c = i$. When the server receives this segment, it also reads its clock counter for a new ISN. However, because there was a crash, the server's clock counter was reset, so that the server reads j again. Therefore, the response segment that the server sends is *(SYN, sn_s, ack_s)*, where $sn_s = j$ and $ack_s = i + 1$. After sending the response, the server receives the last retransmission of the *(sn_c, ack_c, msg_c)* segment, where $sn_c = i + 1$, $ack_c = j + 1$, and $msg_c = m$. The server can accept this data and pass it to the user. Thus, we have duplicate delivery of the same data.

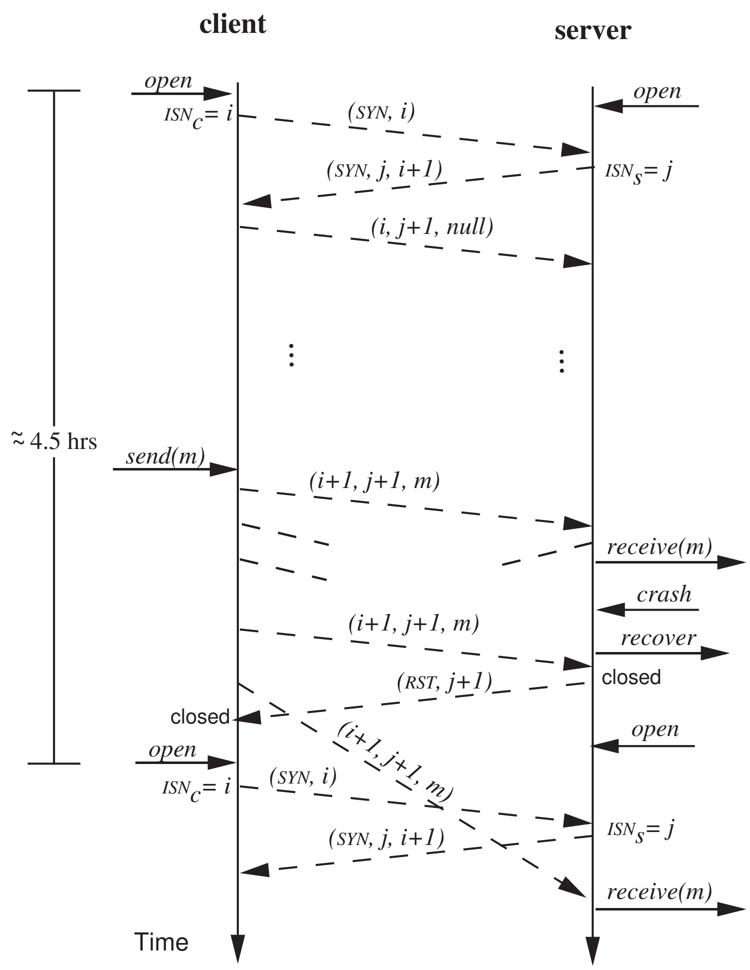

Figure 8-2: ***Example two.*** *Another execution where a long-lived incarnation results in the delivery of the same message twice.*

The scenario in the second execution does not occur if a TCP host times out after a duration of wt when it sends a segment with data and does not receive an acknowledgment, and if quiet time is extended. The extension of quiet time to qt and the maximum wait timeout means that when the server crashes in the above scenario, it does not recover until all copies of the $(i+1,\ j+1,\ m)$ segment have drained from the channel.

The third example demonstrates why after a timeout a host must wait for a period of μ before it closes. This execution starts out like example two, but instead of the server crashing and the client closing because of a reset, they both close because the maximum wait time has elapsed. However, right before it times out, the client sends another copy of the $(i+1,\ j+1,\ m)$ segment that does not get dropped. After it closes the client

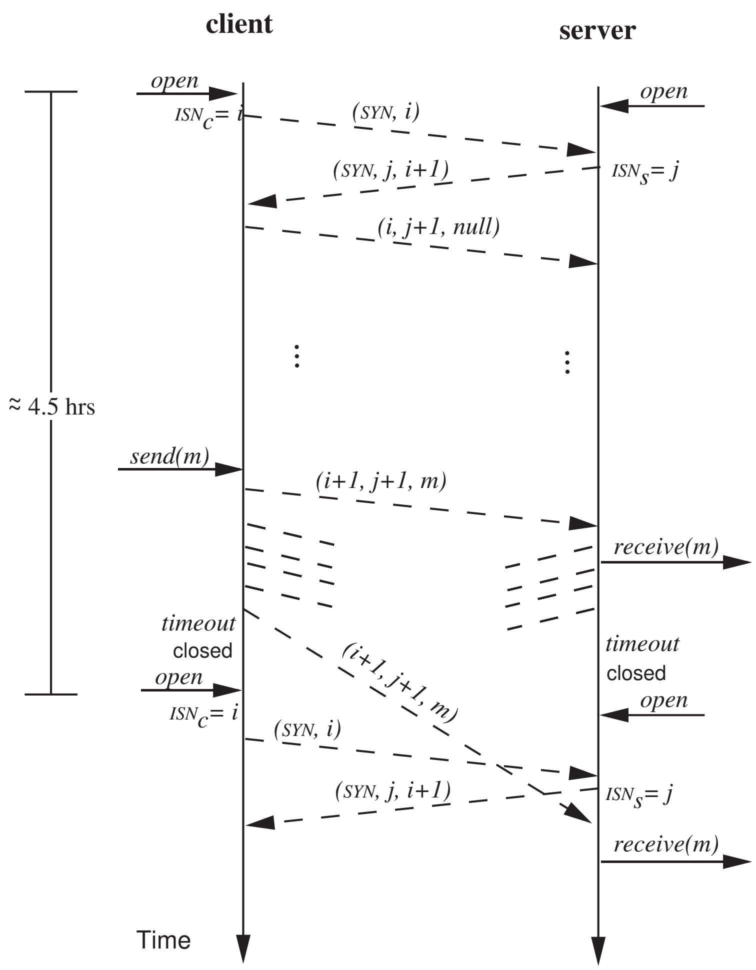

Figure 8-3: ***Example three.*** *This execution demonstrates why a wait of μ is needed after timeouts.*

immediately gets the signal to open and sends a SYN segment with sequence number i. The server also receives the signal to open, and when it receives the SYN segment from the client reads ISN j from the clock counter. The server acknowledges the SYN segment of the client and sends its own SYN with the *(SYN, j, $i+1$)* segment. After sending this segment, the server receives the duplicate copy of the *($i+1$, $j+1$, m)* segment sent right before the client timed out. The server can accept this segment and can pass m to its user again. For this scenario, the duplicate delivery does not occur if the client waits for a period of μ after the timeout before it closes, so that when it reopens, any segment sent before the timeout has been dropped from the channel because of the MSL.

The fourth and final example we present does not involve a long-lived connection, but shows how a crash may cause data from a previous incarnation to be accepted in a current incarnation. It is shown in Figure 8-4. In this example the client and server start with the three-way handshake as in the previous example. After the client sends the third segment in three-way handshake, it immediately crashes. This segment takes time μ to reach the server. Therefore, immediately after the segment arrives at the server the client can send a recover output and reopen. Right before the server receives this segment, it sends a final retransmission of the *(SYN, j, $i+1$)* segment. When the client reopens, because of the crash it reads i from the clock counter again and sends the *(SYN, i)* segment again. This segment is dropped from the channel, but after this segment is sent, the client receives the *(SYN, j, $i+1$)* segment that was sent by the server. When the client receives this segment it sends an acknowledgment which also gets dropped. However, after it sends the acknowledgment, it receives a segment from the server with sequence number $j+1$ and data m. The server sent this segment after it received the acknowledgment the client sent from the previous incarnation. Because this segment has sequence number $j+1$, the client accepts it and can pass message m to the user.

This situation does not happen if the period after the crash is long enough. The problem that occurred with this execution is that a response to a segment sent before the crash is still in the channel after the crashed host recovers. If a host crashes immediately after sending a segment, that segment can take time μ before it arrives. After it arrives, it may take the receiving host rt to respond, and it might retransmit the response segment for a period of wt. The last retransmitted segment may take time μ to arrive. Therefore, after a host crashes, a segment that is a response to a segment sent before the host crashed might be in the channel up to time $2\mu + wt + rt$ after the crash. Hence we set quiet time such that $qt > 2\mu + wt + rt$.

8.3 The formal model

To formally model TCP with bounded counters and the additional timeout mechanisms needed to ensure correctness, we make some changes to the $\mathcal{TCP}$ automaton. The new client side and server side automata are $\mathcal{BTCP}_c$ and $\mathcal{BTCP}_s$ respectively.

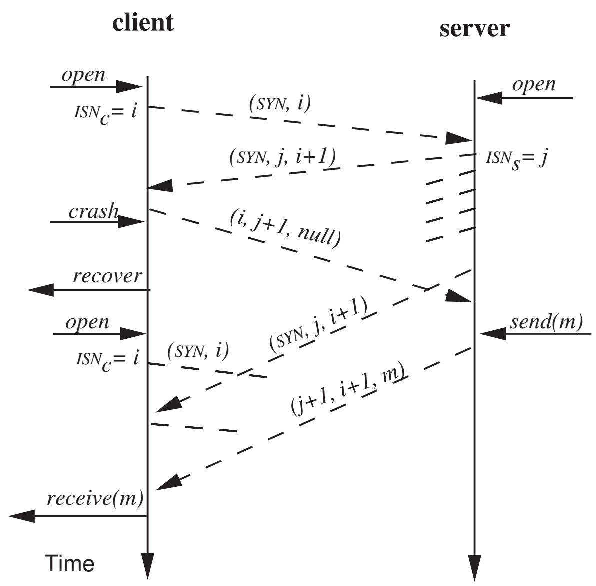

Figure 8-4: ***Example four.*** *An execution where a crash causes a message from a previous incarnation to be delivered in the current incarnation.*

The action signature for $\mathcal{BTCP}_c$ remains the same as for $\mathcal{TCP}_c$ except for the addition of the internal action *clock-counter-tick*$_c$. This internal action increments the client's clock counter. Similarly the action signature of $\mathcal{BTCP}_s$ is the same as it is for $\mathcal{TCP}_s$, but with the addition of the internal action *clock-counter-tick*$_s$, which increments the server's clock counter.

8.3.1 States and start states

The set of states and start states are different because $\mathcal{BTCP}_c$ and $\mathcal{BTCP}_s$ have some additional variables not in $\mathcal{TCP}$ and the type of some variables also change. While not stable, the clock counter variables *clock-counter*$_c$ and *clock-counter*$_s$ are different from the other variables in that when the respective hosts are closed, they are not deleted, and are not reset when the hosts reopen to the initial values given in the table. All the other variables in the table are undefined when the connection is closed, and take on the initial values in the table when TCB_c or TCB_s is initialized. A full description of the changed and new variables is given in the tables below. All other variables remain the same as they are for $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$. The first table summarizes the different types used, and the second table summarizes the constants used.

Type definitions

Type	Description
Msg	The set of all possible strings over some basic message alphabet that does not include the special symbol `null`.
`T`	The nonnegative real numbers — represents real time.
`BN`	The range of the bounded counters — $\{i \mid 0 \leq i < 2^{32}\}$.

Constants

Constant	Description
μ	Maximum Segment Liftime — 2 minutes.
rt	Maximum response time — 0.5 seconds.
wt	Maximum time to wait for a response — 5 minutes.
qt	Quiet time after a crash — $qt > 2\mu + rt + wt$.
ct	The cycle time of the clock counter — 4.5 hours.
clock-rate	The speed at which the clock counters change — $4.5/2^{32}$.
data-rate	The maximum speed at which host can increment its sequence number — $(wt + rt + 2\mu)/2^{32}$.

Client variables

Variable	Type	S	Initially	Description
$clock\text{-}counter_c$	`BN`		`nil`	The clock counter that the client reads for initial sequence numbers.
sn_c	$\mathtt{BN} \cup \mathtt{nil}$		`nil`	The seqence number of the client.
ack_c	$\mathtt{BN} \cup \mathtt{nil}$		`nil`	The acknowledgment number at the client.
$wait\text{-}t\_o_c$	$\mathtt{T} \cup \infty$		∞	Used to mark the time after which the client will stop sending a segment if it does not get a response to that segment.
$last(response_c)$	`T`		0	The upper bound on when the client must send a response to an input that needs one.
$first(reset_c)$	$\mathtt{T} \cup \infty$		∞	The lower bound on when the client can close after receiving a valid reset.
$first(recov_c)$	$\mathtt{T} \cup \infty$		∞	The lower bound on the time when the client can recover after crashing.
$first(prep\text{-}msg_c)$	`T`		*data-rate*	The lower bound on when the client can next prepare a message to be sent.
$first(tick_c)$	`T`	$\surd$	*clock-rate*	The lower bound on when the clock counter can be incremented.
$last(tick_c)$	`T`	$\surd$	*clock-rate*	The upper bound on when the clock counter should be incremented.
$first(open_c)$	`T`	$\surd$	0	The lower bound on the time when the client can open after it closes.

Server variables

Variable	Type	S	Initially	Description
clock-counter$_s$	BN		nil	The clock counter that the server reads for initial sequence numbers.
sn_s	BN $\cup$ nil		nil	The seqence number of the server.
ack_s	BN $\cup$ nil		nil	The acknowledgment number at the server.
wait-t_o$_s$	T $\cup\, \infty$		∞	Symmetric to *wait-t_o*$_c$.
last(response$_s$*)*	T $\cup\, \infty$		∞	Symmetric to *last(response*$_c$*)*.
first(reset$_s$*)*	T $\cup\, \infty$		∞	Symmetric to *first(reset*$_c$*)*.
first(recov$_s$*)*	T $\cup\, \infty$		∞	Symmetric to *first(recov*$_c$*)*.
first(prep-msg$_s$*)*	T		*data-rate*	Symmetric to *first(prep-msg*$_c$*)*.
first(tick$_s$*)*	T	$\surd$	*clock-rate*	Symmetric to *first(tick*$_c$*)*.
last(tick$_s$*)*	T	$\surd$	*clock-rate*	Symmetric to *last(tick*$_c$*)*.
first(open$_s$*)*	T	$\surd$	0	Symmetric to *first(open*$_c$*)*.

8.3.2 Steps

Several steps of $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ have to be changed to get the correct behavior for $\mathcal{BTCP}_c$ and $\mathcal{BTCP}_s$. We include only the steps of $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ that are changed. In the changed steps we outline the if-then-else statements, but omit the assignments to the original variables of $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ (indicated by ...) unless they change in $\mathcal{BTCP}_c$ or $\mathcal{BTCP}_s$. The steps of $\mathcal{BTCP}_c$ and $\mathcal{BTCP}_s$ that are different from the steps with the same action in $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$ are shown in Figures 8-5, 8-6, and 8-7.

The first changes occur for steps with the *send-msg*$_c$*(open, m, close)* action. Firstly, the client is not allowed to open unless the current time is greater than or equal to *first(open*$_c$*)*. This variable is initially 0, but is set to the current time plus the clock rate whenever the client closes. This means at least one clock tick must occur before the client is allowed to reopen after closing. Steps with the *passive-open* action on the server side have the same restriction. Also new in the *send-msg*$_c$*(open, m, close)* step, is that instead of incrementing the sn_c variable to get a new initial sequence number, the number is read from the clock based counter *clock-counter*$_c$. The other change in this action is that the new variable *last(response*$_c$*)* is assigned to $now_c + rt$. This assignment forces the client to perform the *send-seg*$_{cs}$*(SYN, sn*$_c$*)* action within time rt of receiving this input because time is not allowed to advance beyond this time, unless *last(response*$_c$*)* is set to ∞ in this step. The lower bound of *last(response*$_c$*)* on when the client must perform the step with the *send-seg*$_{cs}$*(SYN, sn*$_c$*)* action coupled with the wt bound that the client sets when it performs the action means that if the client does not get a response to the SYN segment within time $wt + rt$ of receiving the *send-msg*$_c$*(open, m, close)* input, it times out.

As mentioned before, the client starts the maximum wait timer in *send-seg*$_{cs}$*(SYN, sn*$_c$*)* step. The test that *wait-t_o*$_c$ is ∞ means the timer is set only when the segment is first sent, and not on every retransmission. To prevent the sending of the segment after the wt period has expired, the action includes in the precondition that $now_c \leq wait\text{-}t\_o_c$. All other actions to send segments on the client side have this precondition also, and the send segment actions on the server side have a symmetric condition. In the *send-seg*$_{cs}$*(SYN, sn*$_c$*)* action, *last(response*$_c$*)* is also set to 0 to indicate that the client has responded to the last input. When the server receives the *receive-seg*$_{cs}$*(SYN, sn*$_c$*)* action, it reads a new initial sequence number from its clock counter and sets its maximum response timer.

The server responds to the *receive-seg*$_{cs}$*(SYN, sn*$_c$*)* action with the *send-seg*$_{sc}$*(SYN, sn*$_s$*, ack*$_s$*)* action. Since it is sending a response, the server resets *last(response*$_s$*)* to ∞ in this action. It also starts the maximum wait timer in this action. When the client receives this segment with the *receive-seg*$_{sc}$*(SYN, sn*$_s$*, ack*$_s$*)* action, if wt time has not elasped since the open input, the sever resets *wait-t_o*$_c$ to ∞, so that a timeout does not occur, and it sets *last(response*$_c$*)* to start the response timer. The client sends a response to the *(SYN, sn*$_s$*, ack*$_s$*)* segment with either the *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* or *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)* actions. In both actions it resets *last(response*$_c$*)* to stop the response timer, and sets the maximum wait timer. In the *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* action, the *ready-to-send*$_c$ variable is checked before the timer is set, because the timer should only be set if the client is sending a segment that needs a response; that is, a segment with valid data.

When the server receives the *(sn*$_c$*, ack*$_c$*, msg*$_c$*)* segment, it sets the response timer, *last(response*$_s$*)*, if the segment needs a response. That is, if $sn_c = ack_s$. The response timer is also set in the *receive-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)* action. In both the *receive-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* and *receive-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)* actions the server also resets *wait-t_o*$_s$ if $ack_c = sn_s + 1$, that is, if the segments have valid acknowledgments. If the this segment causes the server to close from mode `last-ack` then *first(open*$_s$*)* is set to the current time plus the clock rate, to ensure that at least one clock tick must happen before the client re-opens.

The *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)* actions are symmetric to their client side counterparts, and the corresponding *receive-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and *receive-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)* actions are symmetric to their server side counterparts.

The *prepare-msg*$_c$ and *prepare-msg*$_s$ actions change to restrict the rate at which a host can send new messages. In the actual protocols the rate depends on network speeds and the rate at which the other hosts responds. In our model the channels are allowed to deliver segments in 0 time, so to model the limitations on the rate at which segments can be sent we include a lower bound on how often the *prepare-msg*$_c$ and *prepare-msg*$_s$ actions can be enable. Thus, in order for the *prepare-msg*$_c$ action to be enabled now_c must be greater than or equal to *first(prep-msg*$_c$*)*, and *first(prep-msg*$_c$*)* is set to now_c + *data-rate* in the step.

The automaton for bounded TCP has two new internal actions that are not present in unbounded TCP, these actions are *clock-counter-tick*$_c$ and *clock-counter-tick*$_s$. The steps with these actions control the ticking of *clock-counter*$_c$ and *clock-counter*$_s$ respectively. The upper and lower bound for the next time the *clock-counter-tick*$_c$ action is enable are both set to now_c + *clock-rate* in this set. These settings mean *clock-counter*$_c$ is incremented every *clock-rate* seconds, if the client is not in recovery mode. The symmetric settings occur in the *clock-counter-tick*$_s$ step.

As mentioned above, in the time-passage action ($\nu(t)$) on both sides , the precondition is changed so that time does not advance beyond the upper bound on when a response should be sent, before the response is sent. Time is also not allowed to pass beyond when the next tick of the clock counters should occur unless there is a crash.

The *time-out*$_c$ and *time-out*$_s$ also change. The preconditions on these step change to reflect the fact that in bounded TCP timeouts do not only occur at the end of `timed-wait` state. There can also be a timeout if the the maximum wait timeout has expired. If the timeouts occur in this situation, the mode of the host is set to `reset` to enable the shut down actions, and the timing variable is set, so that there is period of inactivity of at least μ before the host close.

In the *receive-seg*$_{sc}$*(RST,* ack_s*, rst-seq*$_s$*)* and *receive-seg*$_{cs}$*(RST,* ack_c*, rst-seq*$_c$*)* actions if the segments are valid reset then the respective timing variables are set, so that there is a period of inactivity of at least μ before the host is allowed it to close via the *shut-down*$_c$ or *shut-down*$_s$ actions. In steps with the *shut-down*$_c$ action, *first(open*$_c$*)* is set to ensure that at least one clock counter tick happens before the client is allowed to re-open. The symmetric assigment happens in steps with the *shut-down*$_s$ action.

The crash and recover actions also change, so that hosts must wait a period of at least qt after a crash before they recover. The recover actions also assign the clock counters an arbitrary value. Because the client closes after *recover*$_c$, *first(open*$_c$*)* is set to the current time plus the clock rate, to ensure that at least one clock tick must happen before the client re-opens. The server side is symmetric.

8.3.3 Specification of the bounded TCP automaton

The specification of the bounded TCP automaton proceeds along the same lines as the specification of $\mathcal{TCP}$. That is, we first define an automaton $\mathcal{BTCP}'$ that is the parallel composition of the client, server and channel automata. However, for bounded TCP we use the channels, defined in Chapter 5, that enforce the maximum segment lifetime. Figure 8-8 shows the composed system. The composed system is formally defined as follows:

$$\mathcal{BTCP}' \triangleq \mathcal{BTCP}_c \| \mathcal{BTCP}_s \| \mu Ch_{cs}(\mathcal{P}) \| \mu Ch_{sc}(\mathcal{P}).$$

To get the user interface that will enable us to show a simulation from the bounded TCP automaton to the abstract specifications, we need to hide the set of actions $\mathcal{A}_T$ defined in Section 6.2. Thus, the general timed automaton model for TCP with bounded counters is defined as:

$$\mathcal{BTCP} \triangleq \mathcal{BTCP}' \backslash \mathcal{A}_T.$$

8.4 Verification of $\mathcal{BTCP}$

$\mathcal{BTCP}$ does not implement $\mathcal{TCP}$ presented it in Chapter 6. It does not implement $\mathcal{TCP}$ because the steps of $\mathcal{BTCP}$ with *time-out*$_c$ or *time-out*$_s$ actions are enabled when certain timeouts expire, whereas in $\mathcal{TCP}$ these steps are only enabled if the respective host is in

send-msg$_c$(open, m, close)
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
  if $mode_c$ = `closed` $\wedge$ *open* $\wedge$
  $now_c > first(open_c)$ then {
    initialize TCB_c
    $sn_c := clock\text{-}counter_c$
    $last(response_c) := now_c + rt$
    $first(prep\text{-}msg_c) := now_c + clock\text{-}rate$
    ...

send-seg$_{cs}$(SYN, sn_c)
Pre: (* Precondition clause from $\mathcal{TCP}_c$ *) $\wedge$
  $now_c \leq wait\text{-}t\_o_c$
Eff: $time\text{-}sent_c := now_c$
  $last(response_c) := \infty$
  if $wait\text{-}t\_o_c = \infty$ then
    $wait\text{-}t\_o_c := now_c + wt$

receive-seg$_{sc}$(SYN, sn_s, ack_s)
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
  if $mode_c$ = `syn-sent` $\wedge$ $ack_s = sn_c + 1$
  then {
    $last(response_c) := now_c + rt$
    $wait\text{-}t\_o_c := \infty$
    ...

send-seg$_{cs}$(sn_c, ack_c, msg_c)
Pre: (* Precondition clause from $\mathcal{TCP}_c$ *) $\wedge$
  $now_c \leq wait\text{-}t\_o_c$
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
  ...
  $last(response_c) := \infty$
  if $ready\text{-}to\text{-}send_c \wedge wait\text{-}t\_o_c = \infty$
    then $wait\text{-}t\_o_c := now_c + wt$

prepare-msg$_c$
Pre: (* Precondition clause from $\mathcal{TCP}_c$ *) $\wedge$
  $now_c \geq first(prep\text{-}msg_c)$
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
  $first(prep\text{-}msg_c) := now_c + data\text{-}rate$
  ...

passive-open
Eff: if $mode_s$ = `closed` $\wedge$
  $now_s > first(open_s)$ then {
    initialize TCB_s
    $mode_s$:= `listen`
  }

receive-seg$_{cs}$ (SYN, sn_c)
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
  if $mode_s$ = `listen` then {
    $sn_s := clock\text{-}counter_s$
    $last(response_s) := now_s + rt$
    $first(prep\text{-}msg_c) := now_s + clock\text{-}rate$
    ...

send-seg$_{sc}$(SYN, sn_s, ack_s)
Pre: (* Precondition clause from $\mathcal{TCP}_s$ *) $\wedge$
  $now_s \leq wait\text{-}t\_o_s$
Eff: $time\text{-}sent_s := now_s$
  $last(response_s) := \infty$
  if $wait\text{-}t\_o_s = \infty$ then
    $wait\text{-}t\_o_s := now_s + wt$

receive-seg$_{cs}$(sn_c, ack_c, msg_c)
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
  ...
  else if $mode_s \neq$ `rec` then {
    if $sn_c = ack_s$ then {
      $last(response_s) := now_s + rt$
      ...
    }
    if $ack_c = sn_s + 1$ then {
      $wait\text{-}t\_o_s := \infty$
      ...
      if $mode_s$ = `last-ack` then {
        $mode_s$:= `closed`
        $first(open_s) := now_s + \mu$
      ...

prepare-msg$_s$
Pre: (* Precondition clause from $\mathcal{TCP}_s$ *) $\wedge$
  $now_s \geq first(prep\text{-}msg_s)$
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
  $first(prep\text{-}msg_c) := now_s + data\text{-}rate$
  ...

Figure 8-5: *Steps of $\mathcal{BTCP}$ that differ from the steps of $\mathcal{TCP}$. The client ($\mathcal{BTCP}_c$) steps are on the left and the server ($\mathcal{BTCP}_s$) steps are on the right.*

receive-seg$_{sc}$*(*sn_s, ack_s, msg_s*)*
Eff: (* Effect clause of $\mathcal{TCP}_c$ *)
  ...
  else if $mode_c \neq$ `rec` then {
    if $sn_s = ack_c$ then {
      $last(response_c) := now_c + rt$
      ...
    }
    if $ack_s = sn_c + 1$ then {
      $wait\text{-}t\_o_c := \infty$
      ...
      if $mode_c =$ `last-ack` then {
        $mode_c :=$ `closed`
        $first(open_c) := now_c + \mu$
        ...

send-seg$_{cs}$*(*sn_c, ack_c, msg_c, *FIN)*
Pre: (* Precondition clause from $\mathcal{TCP}_c$ *) $\wedge$
  $now_c \leq wait\text{-}t\_o_c$
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
  $last(response_c) := \infty$
  if $wait\text{-}t\_o_c = \infty$ then
    $wait\text{-}t\_o_c := now_c + wt$
  ...

receive-seg$_{sc}$*(*sn_s, ack_s, msg_s, *FIN)*
Eff: (* Effect clause from $\mathcal{TCP}_c$ *)
  ...
  else if $mode_c \neq$ `rec` then {
    if $sn_s \geq ack_c$ then {
      $last(response_c) := now_c + rt$
      ...
    }
    if $ack_s = sn_c + 1$ then {
      $wait\text{-}t\_o_c := \infty$
      ...

clock-counter-tick$_c$
Pre: $mode_c \neq$ `rec` $\wedge$ $now_c \geq first(tick_c)$
Eff: $clock\text{-}counter_c := clock\text{-}counter_c + 1$
  $first(tick_c) := now_c + clock\text{-}rate$
  $last(tick_c) := now_c + clock\text{-}rate$

send-seg$_{sc}$*(*sn_s, ack_s, msg_s*)*
Pre: (* Precondition clause of $\mathcal{TCP}_s$ *) $\wedge$
  $now_s \leq wait\text{-}t\_o_s$
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
  ...
  $last(response_s) := \infty$
  if $ready\text{-}to\text{-}send_s \wedge wait\text{-}t\_o_s = \infty$
    then $wait\text{-}t\_o_s := now_s + wt$

receive-seg$_{cs}$*(*sn_c, ack_c, msg_c, *FIN)*
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
  ...
  else if $mode_s \neq$ `rec` then {
    if $sn_c \geq ack_s$ then {
      $last(response_s) := now_s + rt$
      ...
    }
    if $ack_c = sn_s + 1$ then {
      $wait\text{-}t\_o_s := \infty$
      ...

send-seg$_{sc}$*(*sn_s, ack_s, msg_s, *FIN)*
Pre: (* Precondition clause of $\mathcal{TCP}_s$ *) $\wedge$
  $now_s \leq wait\text{-}t\_o_s$
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
  $last(response_s) := \infty$
  if $wait\text{-}t\_o_s = \infty$ then
    $wait\text{-}t\_o_s := now_s + wt$
  ...

clock-counter-tick$_s$
Pre: $mode_s \neq$ `rec` $\wedge$ $now_s \geq first(tick_s)$
Eff: $clock\text{-}counter_s := clock\text{-}counter_s + 1$
  $first(tick_s) := now_s + clock\text{-}rate$
  $last(tick_s) := now_s + clock\text{-}rate$

Figure 8-6: *Other steps of* $\mathcal{BTCP}_c$ *and* $\mathcal{BTCP}_s$ *that differ from their counterparts in* $\mathcal{TCP}$.

<table>
<tr>
<td>

$\nu(t)$ (time-passage)
Pre: $t \in R^+ \wedge$
$now_c + t \leq last(response_c) \wedge$
$((mode_c \neq \texttt{rec} \wedge now_c + t \leq last(tick_s))$
$\vee\ mode_c = \texttt{rec})$
Eff: $now_c := now_c + t$

$time\text{-}out_c$
Pre: $mode_c \notin \{\texttt{rec}, \texttt{reset}, \texttt{closed}\} \wedge$
$(mode_c = \texttt{timed-wait} \wedge$
$now_c \geq first(t\text{-}out_c)) \vee$
$(now_c \geq wait\text{-}t\_o_c)$
Eff: if $(mode_c = \texttt{timed-wait} \wedge$
$(now_c \geq first(t\text{-}out_c))$ then {
$mode_c := \texttt{closed}$
}
else $mode_c := \texttt{reset}$

$receive\text{-}seg_{sc}(RST,\ ack_s,\ rst\text{-}seq_s)$
Eff: if $mode_c \neq \texttt{rec} \wedge rst\text{-}seq_s = ack_c \vee$
$(rst\text{-}seq_s = 0 \wedge ack_s = sn_c + 1)$ then
$mode_c := \texttt{reset}$

$shut\text{-}down_c$
Pre: $mode_c = \texttt{reset}$
Eff: $mode_c := \texttt{closed}$
$first(open_c) := now_c + \mu$

$crash_c$
Eff: if $mode_c \neq \texttt{closed}$ then
$mode_c := \texttt{rec}$
$first(recov_c) := now_c + qt$
$first(open_c) := now_c + qt$
$recover_c$
Pre: $mode_c = \texttt{rec} \wedge$
$now_c \geq first(recov_c)$
Eff: $mode_c := \texttt{closed}$
$clock\text{-}counter_c :\in \texttt{BN}$
$first(tick_c) := now_c + clock\text{-}rate$
$last(tick_c) := now_c + clock\text{-}rate$

</td>
<td>

$\nu(t)$ (time-passage)
Pre: $t \in R^+ \wedge$
$now_s + t \leq last(response_s) \wedge$
$((mode_s \neq \texttt{rec} \wedge now_s + t \leq last(tick_s))$
$\vee\ mode_s = \texttt{rec})$
Eff: $now_s := now_s + t$

$time\text{-}out_s$
Pre: $mode_s \notin \{\texttt{rec}, \texttt{reset}, \texttt{closed}\} \wedge$
$(mode_s = \texttt{timed-wait} \wedge$
$now_s \geq first(t\text{-}out_s)) \vee$
$(now_s \geq wait\text{-}t\_o_s)$
Eff: if $(mode_s = \texttt{timed-wait} \wedge$
$(now_s \geq first(t\text{-}out_s))$ then {
$mode_s := \texttt{closed}$
}
else $mode_s := \texttt{reset}$

$receive\text{-}seg_{cs}(RST,\ ack_c,\ rst\text{-}seq_c)$
Eff: if $mode_s \neq \texttt{rec} \wedge rst\text{-}seq_c = ack_s$ then
$mode_s := \texttt{reset}$

$shut\text{-}down_s$
Pre: $mode_s = \texttt{reset}$
Eff: $mode_s := \texttt{closed}$
$first(open_s) := now_s + \mu$

$crash_s$
Eff: if $mode_s \neq \texttt{closed}$ then
$mode_s := \texttt{rec}$
$first(recov_s) := now_s + qt$
$first(open_s) := now_s + qt$
$recover_s$
Pre: $mode_s = \texttt{rec} \wedge$
$now_s \geq first(recov_s)$
Eff: $mode_s := \texttt{closed}$
$clock\text{-}counter_s :\in \texttt{BN}$
$first(tick_s) := now_s + clock\text{-}rate$
$last(tick_s) := now_s + clock\text{-}rate$

</td>
</tr>
</table>

Figure 8-7: *The remaining steps of $\mathcal{BTCP}_c$ and $\mathcal{BTCP}_s$ that differ from their counterparts in $\mathcal{TCP}$.*

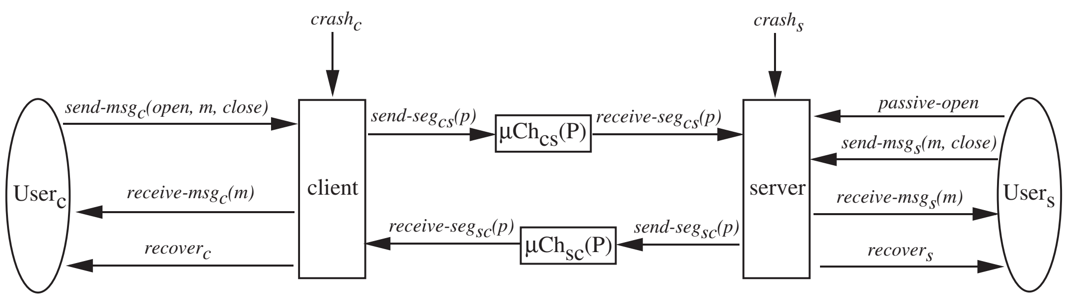

Figure 8-8: *The different components of TCP with bounded counters*

`timed-wait` state. Therefore, these steps of $\mathcal{BTCP}$ cannot be simulated by $\mathcal{TCP}$. That is, $\mathcal{BTCP}$ closes in situations where $\mathcal{TCP}$ does not. However, we can change $\mathcal{TCP}$ slightly so that $\mathcal{BTCP}$ implements this slightly modified version of the protocol. We describe the changes to $\mathcal{TCP}$ in the next section. This new version of $\mathcal{TCP}$ must still implement *patient(S)* in order for us to conclude that $\mathcal{BTCP}$ implements the specification if we show it implements the new version of TCP.[2] We prove that it does in the next section. We define the modified version of $\mathcal{TCP}$ and show that $\mathcal{BTCP}$ implements this protocol because it is easier to show than directly showing a simulation relation from $\mathcal{BTCP}$ to the abstract specification.

8.4.1 Non-deterministic TCP

The change we make to $\mathcal{TCP}$ is simple. We add two internal actions $\textit{set-reset}_c$ and $\textit{set-reset}_s$ which non-deterministicically sets $\textit{mode}_c$ and $\textit{mode}_s$ respectively to `reset`. We call this version of TCP, $\mathcal{NTCP}$, and with the same history variables as $\mathcal{TCP}^h$ we call it $\mathcal{NTCP}^h$. The new steps with the actions $\textit{set-reset}_c$ and $\textit{set-reset}_s$ are shown in Figure 8-9. To show that $\mathcal{NTCP}$ implements *patient(S)*, we need to show that the $\textit{set-reset}_c$ and $\textit{set-reset}_s$ steps can be simulated in D^p. That is, we prove the following lemma:

Lemma 8.1

$\mathcal{NTCP}^h \leq^t_R D^p$ *via* R_{TD}.

Proof: The proof for this lemma is the same as the proof for Lemma 7.1 except we need to add prove of correspondence for the cases with $a = \textit{set-reset}_c$ and $a = \textit{set-reset}_s$. We only

[2] Recall that in Chapter 3 we defined a *patient* version of an untimed automaton to be one where arbitrary time passage steps are added.

set-reset$_c$	*set-reset*$_s$
`Pre`: $mode_c \notin \{\texttt{rec}, \texttt{closed}\}$	`Pre`: $mode_s \notin \{\texttt{rec}, \texttt{closed}\}$
`Eff`: $mode_c := \texttt{reset}$	`Eff`: $mode_s := \texttt{reset}$

Figure 8-9: *The new set-reset steps for* $\mathcal{NTCP}$.

show the case for a = *set-reset*$_c$ since the proof for *set-reset*$_s$ is symmetric.

a = *set-reset*$_c$.

For this step the corresponding α is *(u, abort*$_c$*, u''', mark*$_c$*(I), u'', drop*$_c$*(I', k), u')*. The proof that this execution fragment preserves the correspondence of states is the same as for Case 2 for the step with a = *receive-seg*$_{sc}$*(RST, ack*$_s$*, rst-seq*$_s$*)* in the verification of $\mathcal{TCP}$ presented in Chapter 7. ■

We can now prove that $\mathcal{NTCP}$, implements a patient version of Specification S. That is, we prove the following theorem:

Theorem 8.1

$\mathcal{NTCP} \sqsubseteq_t patient(S)$.

Proof: The proof is essentially identical to the proof of Theorem 7.1. From Lemma 8.1 we get that $\mathcal{NTCP}^h \leq_R^t D^p$, which because of the soundness of timed refinement mapping (Theorem 3.6) and the soundness of adding history variables (Theorem 3.9) implies that $\mathcal{NTCP} \sqsubseteq_t D^p$. From Theorem 4.1 we know $D \sqsubseteq S$. Using the *Embedding Theorem* of [31] presented in Chapter 3 we now get $D^p \sqsubseteq_t patient(S)$. Thus, we now have $\mathcal{NTCP} \sqsubseteq_t D^p$ and $D^p \sqsubseteq_t patient(S)$. Therefore, since the subset relation and thus the implements relation is transitive we get $\mathcal{NTCP} \sqsubseteq_t patient(S)$. ■

Derived variables for $\mathcal{NTCP}$

To verify the correctness of $\mathcal{BTCP}$ we will show a timed forward simulation from the states of $\mathcal{BTCP}$ to $\mathcal{NTCP}$. To facilitate the description of the timed forward simulation, we define a set of derived variables for $\mathcal{NTCP}$. The first two variables *max-sn*$_{cs}$, and *max-sn*$_{sc}$ are the maximum of all the sequence numbers for segments in *in-transit*$_{cs}$ and *in-transit*$_{sc}$ respectively. Similarly we define *max-ack*$_{cs}$, and *max-ack*$_{sc}$ to be the maximum of all the acknowledge numbers for segments in $in\text{-}transit_{cs}$ and $in\text{-}transit_{sc}$ respectively. Let s be any state of $\mathcal{NTCP}$, then these variables are formally defined as follows:

$$s.max\text{-}sn_{cs} \triangleq \begin{cases} \max(sn(p), \text{for all } p \in s.in\text{-}transit_{cs}) & \text{if } s.in\text{-}transit_{cs} \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

$$s.max\text{-}sn_{sc} \triangleq \begin{cases} \max(sn(p), \text{for all } p \in s.in\text{-}transit_{sc}) & \text{if } s.in\text{-}transit_{sc} \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

$$s.max\text{-}ack_{cs} \triangleq \begin{cases} \max(ack(p), \text{for all } p \in s.in\text{-}transit_{cs}) & \text{if } s.in\text{-}transit_{cs} \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

$$s.max\text{-}ack_{sc} \triangleq \begin{cases} \max(ack(p), \text{for all } p \in s.in\text{-}transit_{sc}) & \text{if } s.in\text{-}transit_{sc} \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

We also define $max\text{-}u\text{-}sn_c$ which is the maximum of $max\text{-}sn_{cs}$ and $max\text{-}ack_{sc} - 1$. This variable represents the maximum sequence number we can deduce the client sent, based solely on information on the channels. We use $max\text{-}ack_{sc} - 1$ because it represents the maximum sequence number acknowledged by a segment from the server that is still in $in\text{-}transit_{sc}$. The symmetric variable is $max\text{-}u\text{-}sn_s$.

$$s.max\text{-}u\text{-}sn_c \triangleq \max(s.max\text{-}sn_{cs},\ s.max\text{-}ack_{sc} - 1).$$
$$s.max\text{-}u\text{-}sn_s \triangleq \max(s.max\text{-}sn_{sc},\ s.max\text{-}ack_{cs} - 1).$$

8.4.2 $\mathcal{BTCP}$ with history variables

Before we define the relation between the states of $\mathcal{BTCP}$ and $\mathcal{NTCP}$, we need to add some history variables to $\mathcal{BTCP}$. We denote $\mathcal{BTCP}$ with history variables as $\mathcal{BTCP}^h$. The history variables are mainly used to facilate proofs of invariants of the state protocol of the protocol. In particular, because there are correctness properties that rely on the rate of the clock counters and the rate at which new segments can be sent, we add history variables that record the last time a the clock counter has a value, and the last time the sequence number variable has a value. These variables are *lst-time-cc*$_c$ and *lst-time-cc*$_s$ for clock counter values, and *lst-time-sn*$_c$ and *lst-time-sn*$_s$ for sequence number values. We also have *lst-time-ack*$_c$ and *lst-time-ack*$_s$ that mark the last time the ack_c and ack_s respectively, have particular values. The times of the last crashes at the client and server are stored in the *lst-crash-time*$_c$ and *lst-crash-time*$_s$ variables respectively. It is also important that when a new initial sequence number is chosen or the sequence number is incremented that these numbers cannot be confused with sequence numbers of segments on the channel. To mark the steps when a new initial sequence number is chosen, or when the sequence number is incremented, we add flags *new-isn*$_c$ and *new-isn*$_s$ when new initial sequence numbers are read, and *new-sn*$_c$ and *new-sn*$_s$, when new sequence numbers are generated. Since long-lived connections are a potential source of problems for TCP with bounded counters, we add a history variables *con-strt-time*$_c$ and *con-strt-time*$_s$ that record the start time of the connection on the client and server side respectively, so the duration of the connection can be calculated. We also have history variables *just-estb* and *ack-from-syn* to indicate when certain events occur. They are used to facilitate the proofs of some invariants of $\mathcal{BTCP}$. The tables below provides more details on the history variables.

Variable	**Type**	**S**	**Initially**	**Description**
lst-time-cc$_c$	an array indexed by `BN`, of `T`.	$\surd$	$\forall\ i \in$ `BN`, *lst-time-cc*$_c(i) = 0$	The last time the client side clock counter has value i.
lst-time-cc$_s$	an array indexed by `BN`, of `T`.	$\surd$	$\forall\ j \in$ `BN`, *lst-time-cc*$_s(j) = 0$	Symmetric to *lst-time-cc*$_c$.
lst-time-sn$_c$	an array indexed by `BN`, of `T`.	$\surd$	$\forall\ i \in$ `BN`, *lst-time-sn*$_c(i) = 0$	The last time sn_c has value i.
lst-time-sn$_s$	an array indexed by `BN`, of `T`.	$\surd$	$\forall\ j \in$ `BN`, *lst-time-sn*$_s(j) = 0$	Symmetric to *lst-time-sn*$_c$.
lst-time-ack$_c$	an array indexed by `BN`, of `T`.	$\surd$	$\forall\ i \in$ `BN`, *lst-time-ack*$_c(i) = 0$	The last time ack_c has value i.
lst-time-ack$_s$	an array indexed by `BN`, of `T`.	$\surd$	$\forall\ j \in$ `BN`, *lst-time-ack*$_s(j) = 0$	Symmetric to *lst-time-ack*$_c$.
new-sn$_c$	Bool		`false`	A flag that is set to true when the client chooses a new sequence number, and is set to false when this sequence number is used.
new-sn$_s$	Bool		`false`	Symmetric to *new-sn*$_c$.

Variable	Type	S	Initially	Description
$new\text{-}isn_c$	Bool		`false`	A flag that is set to true when the client chooses a new initial sequence number, and is set to false when this sequence number is used.
$new\text{-}isn_s$	Bool		`false`	Symmetric to $new\text{-}isn_c$.
$con\text{-}strt\text{-}time_c$	`T`		∞	The time when the client chooses an intial sequence number.
$con\text{-}strt\text{-}time_s$	`T`		∞	The time when the server chooses an initial sequence.
$lst\text{-}crash\text{-}time_c$	`T`	$\surd$	0	The time of the last crash at the client.
$lst\text{-}crash\text{-}time_s$	`T`	$\surd$	0	Symmetric to $lst\text{-}crash\text{-}time_c$.
$ack\text{-}from\text{-}syn$	Bool $\cup$ `nil`		`nil`	A flag that is set to true when the client sets its acknowledgment number based on a SYN segment form the server and set to false when the acknowledgment number is base on a non-SYN segment.

Steps of $\mathcal{BTCP}^h$

In Figures 8-10 and 8-11 we show the steps where $\mathcal{BTCP}^h$ differs from $\mathcal{BTCP}$. As always we omit the assignments to the original variables of $\mathcal{BTCP}$ (again indicated by ...) but outline the if-then-else statements. The first addition is to the *send-msg*$_c$*(open, m, close)* step. In this step the $con\text{-}strt\text{-}time_c$ is assigned to the current time, $new\text{-}isn_c$ is assigned to `true` to indicate that a new sequence number is assigned in this step, and $lst\text{-}time\text{-}sn_c(sn_c)$ is set to now_c to record the time sn_c gets the value read from $clock\text{-}counter_c$. The $new\text{-}isn_c$ variable is set to `false` in the *send-seg*$_{cs}$ *(SYN, sn_c)* step.

When the server performs the *receive-seg*$_{cs}$ *(SYN, sn_c)* step, it sets it's version of the connection start time, $con\text{-}strt\text{-}time_s$ to the current time, assigns $new\text{-}isn_s$ to `true`, and $lst\text{-}time\text{-}sn_s(sn_s)$ to the current time. The server assigns $new\text{-}isn_s$ to `false`, when it performs the *send-seg*$_{sc}$*(SYN, sn_s, ack_s)* action. When the client receives the segment via the *receive-seg*$_{sc}$*(SYN, sn_s, ack_s)* action it sets *just-estb* to `true` and *ack-from-syn* to `true`. $lst\text{-}time\text{-}ack_c(ack_c)$ to the current time.

In the *prepare-msg*$_c$ step, the client assigns $new\text{-}sn_c$ to `true`, and the server assigns $new\text{-}sn_s$ to `true` in the *prepare-msg*$_s$ step. These assigns are make because the client increments its sequence number in the the *prepare-msg*$_c$ step, and the server increments its sequence number in the *prepare-msg*$_s$ step. In the the *prepare-msg*$_c$ step *just-estb* is set to false because this may be the next set the client performs after it gets to mode estb. set to

After the client or server increments their respective sequence numbers in the *prepare-msg*$_c$ or *prepare-msg*$_s$ steps, they are enabled to send segments. When they send these segments (*send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* and *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)* for the client and the symmetric steps for the server) *new-sn*$_c$ and *new-sn*$_s$ are set to false respectively.

When either host receives a segment with valid data, it increments the value of its acknowledgment variable. The time that the acknowledgment variable gets the new value is recorded in the *lst-time-ack*$_c$ history variable on the client side and the *lst-time-ack*$_s$ variable on the server side.

In the *clock-counter-tick*$_c$ and *recover*$_c$ steps the server assigns *lst-time-cc*$_c$(*clock-counter*$_c$) to the current time to record the last time the clock counter has a particular value. The *clock-counter-tick*$_s$ and *recover*$_s$ steps are symmetric on the server side.

Derived variables for $\mathcal{BTCP}^h$

We also define two derived variables for $\mathcal{BTCP}^h$. They are *con-duration*$_c$ and *con-duration*$_s$, and they represent the connection duration from the prespective of the client and server respectively. We formally defined them below.

$$s.con\text{-}duration_c \triangleq \begin{cases} s.now - s.con\text{-}strt\text{-}time_c & \text{if } s.mode_c \neq \texttt{closed}, \\ 0 & \text{otherwise.} \end{cases}$$

$$s.con\text{-}duration_s \triangleq \begin{cases} s.now - s.con\text{-}strt\text{-}time_s & \text{if } s.mode_s \notin \{\texttt{closed}, \texttt{listen}\}, \\ 0 & \text{otherwise.} \end{cases}$$

send-msg$_c$*(open, m, close)*
`Eff`: (* Effect clause from $\mathcal{BTCP}_c$ *)
 if $mode_c =$ `closed` $\wedge$ *open* then {
  $con\text{-}strt\text{-}time_c := now_c$
  $new\text{-}isn_c :=$ `true`
  $lst\text{-}time\text{-}sn_c(sn_c) := now_c$
 ...

send-seg$_{cs}$ *(SYN,* sn_c*)*
`Pre`: (* Precondition clause from $\mathcal{BTCP}_c$ *)
`Eff`: (* Effect clause from $\mathcal{BTCP}_c$ *)
 $new\text{-}isn_c :=$ `false`
 ...

receive-seg$_{sc}$*(SYN,* sn_s, ack_s*)*
`Pre`: (* Precondition clause from $\mathcal{BTCP}_c$ *)
`Eff`: (* Effect clause from $\mathcal{BTCP}_c$ *)
 ...
  if $mode_c =$ `syn-sent` then {
   ...
   $just\text{-}estb :=$ `true`
   $ack\text{-}from\text{-}syn :=$ `true`
   $ack_c := sn_s + 1$
   $lst\text{-}time\text{-}ack_c(ack_c) := now_c$
  }

send-seg$_{cs}$*(*sn_c, ack_c, msg_c*)*
`Pre`: (* Precondition clause from $\mathcal{BTCP}_c$ *)
`Eff`: (* Effect clause from $\mathcal{BTCP}_c$ *)
 $just\text{-}estb :=$ `false`
 $new\text{-}sn_c :=$ `false`
 ...

prepare-msg$_c$
`Pre`: (* Precondition clause from $\mathcal{TCP}_c$ *)
`Eff`: (* Effect clause from $\mathcal{TCP}_c$ *)
 $new\text{-}sn_c :=$ `true`
 $just\text{-}estb :=$ `true`
 if $send\text{-}buf_c \neq \epsilon$ then {
  ...
 }
 if $rcvd\text{-}close_c \wedge send\text{-}buf_c = \epsilon$ then {
  ...
 }
 $lst\text{-}time\text{-}sn_c(sn_c) := now_c$

receive-seg$_{cs}$ *(SYN,* sn_c*)*
`Eff`: (* Effect clause from $\mathcal{BTCP}_s$ *)
 if $mode_s =$ `listen` then {
  ...
  $con\text{-}strt\text{-}time_s := now_s$
  $new\text{-}isn_s :=$ `true`
  $lst\text{-}time\text{-}sn_s := now_c$
  $lst\text{-}time\text{-}ack_s := now_c$
  ...

send-seg$_{sc}$*(SYN,* sn_s, ack_s*)*
`Pre`: (* Precondition clause from $\mathcal{BTCP}_s$ *)
`Eff`: (* Effect clause from $\mathcal{BTCP}_s$ *)
 $new\text{-}isn_s :=$ `false`
 ...

receive-seg$_{cs}$*(*sn_c, ack_c, msg_c*)*
`Eff`: (* Effect clause from $\mathcal{BTCP}_s$ *)
 ...
  if $sn_c = ack_s$ then {
   $ack_s := sn_c + 1$
   $lst\text{-}time\text{-}ack_s(ack_s) := now_s$
   ...

prepare-msg$_s$
`Pre`: (* Precondition clause from $\mathcal{TCP}_s$ *)
`Eff`: (* Effect clause from $\mathcal{TCP}_s$ *)
 $new\text{-}sn_s :=$ `true`
 if $send\text{-}buf_s \neq \epsilon$ then {
  ...
 }
 if $rcvd\text{-}close_s \wedge send\text{-}buf_s = \epsilon$ then {
  ...
 }
 $lst\text{-}time\text{-}sn_s(sn_s) := now_s$

Figure 8-10: *Steps where* $\mathcal{BTCP}^h$ *differs from* $\mathcal{BTCP}$.

$receiv\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$
Eff: (* Effect clause from $\mathcal{BTCP}_c$ *)
. . .
if $sn_s = ack_c$ then {
$ack_c := sn_c + 1$
$lst\text{-}time\text{-}ack_c(ack_c) := now_c$
$ack\text{-}from\text{-}syn :=$ `false`
. . .

$send\text{-}seg_{sc}(sn_c,\ ack_c,\ msg_c,\ FIN)$
Pre: (* Precondition clause from $\mathcal{BTCP}_c$ *)
Eff: (* Effect clause from $\mathcal{BTCP}_c$ *)
$new\text{-}sn_c :=$ `false`
$just\text{-}estb :=$ `false`
. . .

$receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$
Eff: (* Effect clause from $\mathcal{BTCP}_c$ *)
. . .
if $sn_s = ack_c + 1$ then
$ack_c := sn_s + 1$
$lst\text{-}time\text{-}ack_c(ack_c) := now_c$
$ack\text{-}from\text{-}syn :=$ `false`
. . .

$clock\text{-}counter\text{-}tick_c$
Pre: $mode_c \neq \texttt{rec} \ \wedge\ now_c \geq first(tick_c)$
Eff: $clock\text{-}counter_c := clock\text{-}counter_c + 1$
$first(tick_c) := now_c + clock\text{-}rate$
$lst\text{-}time\text{-}cc_c(clock\text{-}counter_c) := now$

$crash_c$
Eff: . . .
$lst\text{-}crash\text{-}time_c := now$

$recover_c$
Pre: $mode_c = \texttt{rec} \ \wedge\ now_c \geq first(recov_c)$
Eff: $mode_c :=$ `closed`
$clock\text{-}counter_c :\in \texttt{BN}$
$lst\text{-}time\text{-}cc_c(clock\text{-}counter_c) := now$

$send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$
Pre: (* Precondition clause from $\mathcal{BTCP}_s$ *)
Eff: (* Effect clause from $\mathcal{BTCP}_s$ *)
$new\text{-}sn_s :=$ `false`
. . .

$receive\text{-}seg_{sc}(sn_c,\ ack_c,\ msg_c,\ FIN)$
Eff: (* Effect clause from $\mathcal{BTCP}_s$ *)
. . .
if $sn_c = ack_s + 1$ then
$ack_s := sn_c + 1$
$lst\text{-}time\text{-}ack_s(ack_s) := now_s$
. . .

$send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$
Pre: (* Precondition clause from $\mathcal{BTCP}_s$ *)
Eff: (* Effect clause from $\mathcal{BTCP}_s$ *)
. . .
$new\text{-}sn_s :=$ `false`

$clock\text{-}counter\text{-}tick_s$
Pre: $mode_s \neq \texttt{rec} \ \wedge\ now_s \geq first(tick_s)$
Eff: $clock\text{-}counter_s := clock\text{-}counter_s + 1$
$first(tick_s) := now_s + clock\text{-}rate$
$lst\text{-}time\text{-}cc_s(clock\text{-}counter_s) := now$

$crash_s$
Eff: . . .
$lst\text{-}crash\text{-}time_s := now$

$recover_s$
Pre: $mode_s = \texttt{rec} \ \wedge\ now_s \geq first(recov_s)$
Eff: $mode_s :=$ `closed`
$clock\text{-}counter_s :\in \texttt{BN}$
$lst\text{-}time\text{-}cc_s(clock\text{-}counter_s) := now$

Figure 8-11: *The other steps where* $\mathcal{BTCP}^h$ *differs from* $\mathcal{BTCP}$.

8.4.3 Invariants

As is the case for $\mathcal{TCP}$, we need to prove a set of invariants on the reachable states of $\mathcal{BTCP}$ in order to limit the states we need to consider for the simulation proof. The proofs for the invariants are presented in Appendix C. In the definition of the invariants of $\mathcal{BTCP}$ and in the proofs, when we talk about a sequence number or acknowledgment number being bigger than another we use the following definition. In $\mathcal{BTCP}$, $i > j$ if and only if $i \in \{j+1, \ldots, j+(2^{31}-1)\}$, where the additions are modulo 2^{32}. The properties stated below are true of all reachable states of $\mathcal{BTCP}^h$.

The first group of invariants, Invariants 8.1 through 8.4, state properties about the relationship between the connection start time, the timestamp on segments, sequence numbers and acknowledgment numbers.

Invariant 8.1 says that when the client is in mode `syn-sent`, and the server is in mode `syn-rcvd`, any segment sent after the respective connection start times has the same sequence number as the current sequence number of the sending host.

Invariant 8.1

1. If $mode_c = \texttt{syn-sent}$ and for $(p,t) \in in\text{-}transit_{cs}$, $t - \mu \geq con\text{-}strt\text{-}time_c$ then $sn_c = sn(p)$.

2. If $mode_s = \texttt{syn-rcvd}$ and for $(p,t) \in in\text{-}transit_{sc}$, $t - \mu \geq con\text{-}strt\text{-}time_s$ then $sn_s = sn(p)$. ∎

Invariant 8.2 states a property that is easy to see. It says that when the client or server chooses a new initial sequence number, any segments on the respective outgoing channels must have been sent before the connection start time (the current time).

Invariant 8.2

1. If $mode_c = \texttt{syn-sent} \wedge new\text{-}isn_c$ then for all $(p,t) \in in\text{-}transit_{cs}$, $t - \mu < con\text{-}strt\text{-}time_c$.

2. If $mode_s = \texttt{syn-rcvd} \wedge new\text{-}isn_s$ then for all $(p,t) \in in\text{-}transit_{sc}$, $t - \mu < con\text{-}strt\text{-}time_s$.

∎

Invariant 8.3 says that any segment on the outgoing channels when a host is not closed must have been sent after the host reopened.

Invariant 8.3

1. If $mode_c \neq \texttt{closed}$ then for all segments $(p, t) \in \textit{in-transit}_{cs}$, $t \geq \textit{con-strt-time}_c + \mu$.

2. If $mode_s \neq \texttt{closed}$ then for all segments $(p, t) \in \textit{in-transit}_{sc}$, $t \geq \textit{con-strt-time}_s + \mu$.

Invariant 8.4 says if a host in a synchronized state receives a segment with data it accepts, and the sending host still has the same sequence number as the sequence number on the segment, then the segment must be from the current incarnation.

Invariant 8.4

1. If $mode_s \in \textit{sync-states}$ and there exists segment $(p, t) \in \textit{in-transit}_{cs}$ such that $sn_c = sn(p)$ and $sn(p) = ack_s$ then $t - \mu \geq \textit{con-strt-time}_c$.

2. If $mode_c \in \textit{sync-states}$ and there exists segment $(p, t) \in \textit{in-transit}_{sc}$ such that $sn_s = sn(p)$ and $sn(p) = ack_c$ then $t - \mu \geq \textit{con-strt-time}_c$. ■

Invariants 8.5 through 8.12 state properties that are true before the hosts become synchronized. These properties are key correctness properties, and together they basically say that the sequence numbers on segments sent during the three-way handshake part of the protocol do not get confused with sequence numbers of old duplicate segments.

Invariant 8.5 says that when the client chooses a new initial sequence number, the other host cannot already have an acknowledgment number that acknowledges that new sequence number. For the client side, the server's acknowledgment number may actually acknowledge the sequence number, but the server cannot be in mode $\texttt{syn-rcvd}$ $with now \leq \textit{wait-t\_o}_s$. That is, the server cannot send a SYN segment with the acknowledgment number. Invariant 8.6 is similar to Invariant 8.5. It states that when the client chooses a new initial sequence numbers, there cannot already be a segment that acknowledges this new sequence number on the incoming channel.

Invariant 8.5

If $mode_c = \texttt{syn-sent} \wedge \textit{new-isn}_c = \texttt{true} \wedge ack_s \in \texttt{BN}$ then $sn_c \geq ack_s \vee \neg(mode_s \neq \texttt{syn-rcvd} \wedge now \leq \textit{wait-t\_o}_s)$.

Invariant 8.6

If $mode_c$ = `syn-sent` $\wedge$ $new\text{-}isn_c$ = `true` then for all SYN segments $(p, t) \in in\text{-}transit_{sc}$, $sn_c \geq ack(p)$.

Invariants 8.7 and 8.8 are symmetric to Invariants 8.5 and 8.6 respectively.

Invariant 8.7

If $mode_s$ = `syn-rcvd` $\wedge$ $new\text{-}isn_s$ = `true` $\wedge$ $ack_c \in$ `BN` then $sn_s \geq ack_c$. ■

Invariant 8.8

If $mode_s$ = `syn-rcvd` $\wedge$ $new\text{-}isn_s$ = `true` then for all segments $(p, t) \in in\text{-}transit_{cs}$, $ack(p) < sn_s + 1$. ■

Invariant 8.9 says that if the client is in mode `syn-sent`, then any segment that acknowledges the sequence number of the client, must contain a sequence number that has not already been acknowledged by a segment already on the channel to the server. In other words, the client has to receive the second segment of the three-way handshake before it can send the third. Invariant 8.10 is along the same lines. It says that the sequence number on the second segment of the three-way handshake must be bigger than the sequence number on any non-SYN segment on the same channel at the same time. That is, the server cannot send a segment with data that the client will accept before the server receives the third segment of the three-way handshake.

Invariant 8.9

If $mode_c$ = `syn-sent` then for all SYN segments $(p, t) \in in\text{-}transit_{sc}$ such that $ack(p) = sn_c + 1$, $sn(p) \geq ack(q)$ for all $(q, t') \in in\text{-}transit_{cs}$. ■

Invariant 8.10

If $mode_c$ = `syn-sent` and there exists SYN segment $(p, t) \in in\text{-}transit_{sc}$ such that $ack(p) = sn_c + 1$ then $sn(p) \geq sn(q)$ for all non-SYN segments $(q, t') \in in\text{-}transit_{sc}$. ■

Invariant 8.11 says that the acknowledgment number of the server when it is in mode `syn-rcvd` is always less than the sequence number of the server plus one when the server is not closed.

Invariant 8.11

If $mode_s = \texttt{syn-rcvd} \ \wedge \ now \leq wait\text{-}t\_o_s \ \wedge mode_c \neq \texttt{closed}$ then $ack_s \leq sn_c + 1$. ■

Informally, Invariant 8.12 says that when the client receives the second segment in the three-way handshake, if the sequence number of the server is such that it can send a segment the client can accept ($sn_s \in \{ack_c, ack_c + 1)$, then the server must be in mode `syn-rcvd`, which means it can only send SYN segments. The client does not accept SYN segments if $mode_c = \texttt{estb}$.

Invariant 8.12

If $just\text{-}estb = \texttt{true} \ \wedge sn_s \in \texttt{BN}$ then $ack_c > sn_s$. ■

Invariants 8.13 through 8.18 deal with the id not-in-use property once the connection has been established. Invariant 8.13 says that once the connection is established, the sequence number at the hosts is at least as big as the sequence number on any out going segment.

Invariant 8.13

1. If $mode_c \in sync\text{-}states$ then for all segments $(p, t) \in in\text{-}transit_{cs}$, $sn_c \geq sn(p)$.

2. If $mode_s \in sync\text{-}states$ then for all segments $(p, t) \in in\text{-}transit_{sc}$, $sn_s \geq sn(p)$. ■

Invariants 8.14 and 8.15 are similar. Invariant 8.14 says that the sequence number plus one at a host in a synchronized state is always greater than or equal to the acknowledgment number at the other host, and Invariant 8.15 says the sequence number plus one is also greater than or equal to the acknowledgment number on any incoming channel.

Invariant 8.14

1. If $mode_c \in sync\text{-}states \ \wedge ack_s \in \texttt{BN}$ then $sn_c + 1 \geq ack_s$.

2. If $mode_s \in sync\text{-}states \ \wedge ack_c \in \texttt{BN}$ then $sn_s + 1 \geq ack_c$. ■

Invariant 8.15

1. If $mode_c \in sync\text{-}states \ \wedge new\text{-}sn_c = \texttt{true}$ then for all segments $(p, t) \in in\text{-}transit_{sc}$, $sn_c + 1 > ack(p)$.

2. If $mode_s \in sync\text{-}states \ \wedge new\text{-}sn_s = \texttt{true}$ then for all segments $(p, t) \in in\text{-}transit_{cs}$, $sn_s + 1 > ack(p)$. ■

Invariant 8.16 says the host are in synchronized states, there acknowledgment numbers are greater than or equal to the acknowledgment number of any segment on the respective outgoing channels.

Invariant 8.16

1. If $mode_c \in sync\text{-}states$ then for all $(p, t) \in in\text{-}transit_{cs}$, $ack_c \geq ack(p)$. ■

2. If $mode_s \in sync\text{-}states$ then for all $(p, t) \in in\text{-}transit_{sc}$, $ack_s \geq ack(p)$. ■

Invariant 8.17 expresses a key correctness property. It states that when a host receives a segment from which it may accept data ($sn(p) \geq ack_c$ or $sn(p) \geq ack_s$), then the sender has not changed its sequence number from the time it sent this segment. Another way to state the property expressed by the invariant is: sequence numbers do not get changed until they are acknowledged.

Invariant 8.17

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states \wedge mode_c \in \{\texttt{rec}, \texttt{reset}\} \cup sync\text{-}states$ and there exists $(p, t) \in in\text{-}transit_{cs}$ such that $sn(p) \geq ack_s$, then $sn_c = sn(p)$.

2. If $mode_c \in sync\text{-}states$ and there exists $(p, t) \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, then $sn_s = sn(p)$. ■

Invariant 8.18 states that when a host receives a segment from which it accepts data, there cannot be another segment on the same channel from which it will accept new data before the old data is acknowledged.

Invariant 8.18

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states$ and there exists $(p, t) \in in\text{-}transit_{cs}$ such that $sn(p) \geq ack_s$, then for all other non-SYN segments $(q, t') \in in\text{-}transit_{cs}$, $sn(q) \leq sn(p)$.

2. If $mode_c \in sync\text{-}states$ and there exists $(p, t) \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, then for all other non-SYN segments $(q, t') \in in\text{-}transit_{sc}$, $sn(q) \leq sn(p)$. ■

The conjunction of all the above invariants is itself an invariant, and we call this invariant I_B.

8.4.4 The Simulation

In this section we define a relation from states of $\mathcal{BTCP}^h$ to states of $\mathcal{TCP}$, and then prove that it is a timed forward simulation with respect to Invariants I_T and I_B.

The timed forward simulation

We define a relation F_{BN} (Definition 8.1) from *states*($\mathcal{BTCP}^h$) to *states*($\mathcal{NTCP}$). Because $\mathcal{NTCP}$ has many variables, we have many cases for the timed forward simulation. The many cases makes the relation look more complicated than it actually is. We discuss the intuition behind relation F_{BN} before we present it, so that it will be easier for the reader to understand the formal definition.

Since $\mathcal{BTCP}^h$ works in the same basic manner as $\mathcal{NTCP}$, the relation defines most of the variables of $\mathcal{BTCP}^h$ to be equal to their counterparts in $\mathcal{NTCP}$. However, since $\mathcal{BTCP}^h$ has a bounded number space for sequence numbers and acknowledgment numbers and $\mathcal{NTCP}$ has an unbounded number space for these variables, we cannot make these variables equal in the relation. Reset numbers are also bounded in $\mathcal{BTCP}^h$, but we are not concerned with these numbers because we will simulate resets in $\mathcal{BTCP}^h$ with *set-reset* actions in $\mathcal{NTCP}$. Since *set-reset*$_c$ and *set-reset*$_s$ are almost always enabled, we do not have to match the settings of the *send-rst*$_c$, *send-rst*$_s$, *rst-seq*$_c$, and *rst-seq*$_s$ variables. Therefore, in the relation the allowable values of these variables in $\mathcal{NTCP}$ is independent of their values in $\mathcal{BTCP}^h$. Basically, these variables never get set in $\mathcal{NTCP}$.

For sequence numbers and acknowledgment numbers, the related values may not be equal because in $\mathcal{BTCP}$ the numbers are generated by bounded counters, while in $\mathcal{NTCP}$ they are generated by an unbounded counter. However, the actual numbers of the individual corresponding variables is not what is important in the protocols, but whether they are equal to, or less than, or greater than the numbers of other variables. This is the key idea in how the relation is defined for these variables.

For the client side sequence number, $u.sn_c$, we have three cases. The first case is always true of the states of $\mathcal{NTCP}$(Invariants 7.1, 7.2 and 7.3). It is the rule used to define the allowable values of $u.sn_c$ if the other two cases are not true, or if $s.mode_c = \texttt{closed}$. In $\mathcal{BTCP}^h$ when a host is closed, the sequence number variable is undefined. In $\mathcal{NTCP}$ sequence numbers are stable, and always have a value. Therefore, when a host is closed in $\mathcal{BTCP}^h$, in the set of related states on $\mathcal{NTCP}$ the sequence number of the host must have a value. We define the set of allowable values to be numbers greater than or equal to the maximum of the maximum sequence number used as reflected by the channels and the acknowledgment number of the other host minus one. The intuition here is that right before a host closes in $\mathcal{BTCP}^h$, its sequence number will at least be this value. The other two cases illustrate the fact that it is whether variables are equal to each other or greater than another variable, and by how much, that is important. The second case is if $s.sn_c = s.ack_s$. This relationship is important because it means the client can send a segment with messages that may be accepted by the server. However, since the client only send messages on non-SYN segments, it is only relevant when the client is in a synchronized state. Also, since the server only accepts messages if it is in a synchronized state it is only relevant if the server is in a synchronized state. The third case for $u.sn_c$ is the situation where the client can send a FIN segment with data. The definition for $u.sn_s$ is essentially symmetric.

The client side acknowledgment number in $\mathcal{NTCP}$, $u.ack_c$, is `nil` or undefined when the corresponding variable in $\mathcal{BTCP}^h$ is `nil`, or undefined. Otherwise, it must be equal to, or less than $u.sn_s+1$ when $s.ack_c$ is less than or equal to $s.sn_s+1$. The important relationship is whether $s.ack_c = s.sn_s + 1$, because the server can send a valid acknowledgment if this is the case. When the sequence number of the server in $\mathcal{BTCP}^h$ is either undefined, or is `nil`, and if the $s.ack_c \neq \texttt{nil}$, then in the related states of $\mathcal{NTCP}$, $u.ack_c \leq u.sn_s + 1$, which we know by Invariant 7.2 is always true in the reachable states of $\mathcal{NTCP}$. For $u.ack_s$ the relation is not quite symmetric. The first non-symmetric aspect is that we specify that whenever $s.mode_s = \texttt{closed}$, then $u.ack_s$ is undefined. We need to specify this here because the server may set $s.ack_s$ to 0, when it generates a reset for a SYN segment received when it is closed. However, since we will simulate resets in $\mathcal{BTCP}^h$ with the *set-reset* actions in $\mathcal{NTCP}$, changes associated with generating reset segments do not change the related values for variables in $\mathcal{NTCP}$. It is also important that if $s.ack_s = s.sn_c+1$ that $u.ack_s = u.sn_c+1$.

However, since only a SYN segment can acknowledge the sequence number of the client when it is in mode `syn-sent`, it only matters that $u.ack_s = u.sn_c + 1$ when the client is not in mode `syn-sent`, or if it is in mode `syn-sent`, that the server is in mode `syn-rcvd`.

We now discuss the relationship between $\mathit{in\text{-}transit}_{cs}$ and $\mathit{in\text{-}transit}_{sc}$ in $\mathcal{BTCP}^h$ and the same variables in $\mathcal{NTCP}$. The relation between these variables is the most complicated part of F_{BN}. However, it is not as complicated as it looks in the formal definition. It looks complicated because it is somewhat difficult to present formally and precisely the relatively simple idea behind this part of the relation. We start with the case for $\mathit{in\text{-}transit}_{cs}$. The basic idea is that each non-RST segment $(p', t') \in s.\mathit{in\text{-}transit}_{cs}$ is related to a segment $p \in u.\mathit{in\text{-}transit}_{cs}$ that has the same message and is the same type (SYN, or FIN, or neither), but there is no timestamp, and the sequence number, and/or the acknowledgment number may be different. As is the case for sequence numbers and acknowledgment numbers, what is important is how the numbers relate to other variables. For sequence numbers of non-RST segments in $s.\mathit{in\text{-}transit}_{cs}$, the related sequence numbers for segments in $u.\mathit{in\text{-}transit}_{cs}$ is determined in a manner similar to how the related values for client side sequence numbers are determined. The first case is always true of the reachable states of $\mathcal{NTCP}$, and is used when the other cases do not hold (Invariant 7.1). The variables the sequence numbers are related to in the relation reflect how the sequence numbers are generated or used in the protocols, so because when segment (p', t') is added to $s.\mathit{in\text{-}transit}_{cs}$ for the first time, $sn(p') = s.sn_c$, in the set of corresponding states $sn(p) = u.sn_c$. However, we do not want sequence numbers of segments from a previous incarnation to be equal to the current sequence number, even if they are in $\mathcal{BTCP}^h$, so we have the added restriction that the timestamp on the segment must indicate that it was placed on the channel after the connection started; that is, $(t - \mu) \geq s.\mathit{con\text{-}strt\text{-}time}_c$. The sequence numbers of segments related to segments from a previous incarnation are always less than $u.sn_c$. Also if the sequence number of the segment in $\mathcal{BTCP}^h$ is actually less than $s.sn_c$, then the sequence number of the related segment is also less than $u.sn_c$. It is also important that if on a non-SYN segment (p', t) that if $sn(p') = s.ack_s$ or $sn(p') = s.ack_s + 1$, the same relation holds for the corresponding segment p. It is important because such a segment has data the server may accept. However, if $s.mode_s =$ `syn-rcvd`, the server only accepts the data if $ack(p') = s.sn_s + 1$.

Acknowledgment numbers are treated in a similar manner. For non-SYN segments it is important that if $ack(p') = s.sn_s + 1$, then $ack(p) = u.sn_s + 1$, because when the server receives a segment it checks if the acknowledgment number has this value. However, since the server does not check if $ack(p') = s.sn_s$ or any value less than that, we do not need to be as precise with the relationship between acknowledgment numbers that are less than or equal to the sequence number of the receiving host. We also have an invariant rule, for when the other conditions do not hold. That is, $ack(p) \leq u.sn_s + 1$.

The relation for $in\text{-}transit_{sc}$ is essentially symmetric, except that SYN segments in this channel have acknowledgment numbers, so they are treated in a slightly different manner. Also, because the server may assign an acknowledgment number based on an old duplicate SYN segment from the client, it is only important that the acknowledgment number of the segment is equal to the acknowledgment number of the server, if the segment was sent after the connection started at the server. We now present the formal definition of F_{BN}.

Definition 8.1 (Forward Simulation from $\mathcal{BTCP}^h$ to $\mathcal{NTCP}$)
If $s \in states(\mathcal{BTCP}^h)$ then define F_{BN} to be the state $u \in states(\mathcal{NTCP})$ such that:

1. $u.now = s.now$
2. $u.ready\text{-}to\text{-}send_c = s.ready\text{-}to\text{-}send_c$
   $u.ready\text{-}to\text{-}send_s = s.ready\text{-}to\text{-}send_s$
3. $u.send\text{-}ack_c = s.send\text{-}ack_c$
   $u.send\text{-}ack_s = s.send\text{-}ack_s$
4. $u.send\text{-}fin_c = s.send\text{-}fin_c$
   $u.send\text{-}fin_s = s.send\text{-}fin_s$
5. $u.push\text{-}data_c = s.push\text{-}data_c$
   $u.push\text{-}data_s = s.push\text{-}data_s$
6. $u.msg_c = s.msg_c$
   $u.msg_s = s.msg_s$
7. $u.send\text{-}fin\text{-}ack_c = s.send\text{-}fin\text{-}ack_c$
   $u.send\text{-}fin\text{-}ack_s = s.send\text{-}fin\text{-}ack_s$
8. $u.time\text{-}sent_c = s.time\text{-}sent_c$
   $u.time\text{-}sent_s = s.time\text{-}sent_s$
9. $u.send\text{-}buf_c = s.send\text{-}buf_c$
   $u.send\text{-}buf_s = s.time\text{-}sent_s$
10. $u.rcv\text{-}buf_c = s.rcv\text{-}buf_c$
    $u.rcv\text{-}buf_s = s.rcv\text{-}buf_s$
11. $u.mode_c = s.mode_c$
    $u.mode_s = s.mode_s$

12. $$\begin{array}{llll} u.\mathit{send\text{-}rst}_c & = & \mathtt{false} & \text{if } s.\mathit{mode}_c \neq \mathtt{closed} \\ & \text{is} & \text{undefined} & \text{if } s.\mathit{mode}_c = \mathtt{closed} \\ u.\mathit{send\text{-}rst}_s & = & \mathtt{false} & \text{if } s.\mathit{mode}_s \neq \mathtt{closed} \\ & \text{is} & \text{undefined} & \text{if } s.\mathit{mode}_s = \mathtt{closed} \end{array}$$

13. $$\begin{array}{llll} u.\mathit{rst\text{-}seq}_c & = & \mathtt{nil} & \text{if } s.\mathit{mode}_c \neq \mathtt{closed} \\ & \text{is} & \text{undefined} & \text{if } s.\mathit{mode}_c = \mathtt{closed} \\ u.\mathit{rst\text{-}seq}_s & = & \mathtt{nil} & \text{if } s.\mathit{mode}_s \neq \mathtt{closed} \\ & \text{is} & \text{undefined} & \text{if } s.\mathit{mode}_s = \mathtt{closed} \end{array}$$

14. $$\begin{array}{llll} u.\mathit{sn}_c & \geq & \max(u.\mathit{max\text{-}u\text{-}sn}_c, u.\mathit{ack}_s - 1) & \text{if } s.\mathit{mode}_c \in \{\mathtt{closed}, \mathtt{syn\text{-}sent}\} \vee s.\mathit{sn}_c \notin \{s.\mathit{ack}_s, s.\mathit{ack}_s + 1\}. \\ & = & u.\mathit{ack}_s & \text{if } s.\mathit{sn}_c = s.\mathit{ack}_s \wedge s.\mathit{mode}_c \notin \{\mathtt{closed}, \mathtt{syn\text{-}sent}\} \wedge (s.\mathit{mode}_s \neq \mathtt{syn\text{-}rcvd} \vee (s.\mathit{mode}_s = \mathtt{syn\text{-}rcvd} \wedge s.\mathit{ack}_c = \mathit{sn}_s + 1)). \\ & = & u.\mathit{ack}_s + 1 & \text{if } s.\mathit{sn}_c = s.\mathit{ack}_s + 1 \wedge s.\mathit{mode}_c \notin \{\mathtt{closed}, \mathtt{syn\text{-}sent}\} \wedge (s.\mathit{mode}_s \neq \mathtt{syn\text{-}rcvd} \vee (s.\mathit{mode}_s = \mathtt{syn\text{-}rcvd} \wedge s.\mathit{ack}_c = \mathit{sn}_s + 1)). \end{array}$$

15. $$\begin{array}{llll} u.\mathit{sn}_s & \geq & \max(u.\mathit{max\text{-}u\text{-}sn}_s, u.\mathit{ack}_c - 1) & \text{if } s.\mathit{mode}_s \in \{\mathtt{closed}, \mathtt{listen}, \mathtt{syn\text{-}rcvd}\} \vee s.\mathit{sn}_s \notin \{s.\mathit{ack}_c, s.\mathit{ack}_c + 1\}. \\ & = & u.\mathit{ack}_c & \text{if } s.\mathit{sn}_s = s.\mathit{ack}_c \wedge s.\mathit{mode}_s \notin \{\mathtt{closed}, \mathtt{listen}, \mathtt{syn\text{-}rcvd}\} \\ & = & u.\mathit{ack}_c + 1 & \text{if } s.\mathit{sn}_s = s.\mathit{ack}_c + 1 \wedge s.\mathit{mode}_s \notin \{\mathtt{closed}, \mathtt{listen}, \mathtt{syn\text{-}rcvd}\} \end{array}$$

16. $$\begin{array}{llll} u.\mathit{ack}_c & = & s.\mathit{ack}_c & \text{if } s.\mathit{ack}_c = \mathtt{nil} \\ & \leq & u.\mathit{sn}_s + 1 & \text{if } s.\mathit{mode}_s \in \{\mathtt{closed}, \mathtt{listen}\} \wedge s.\mathit{ack}_c \neq \mathtt{nil} \\ & = & u.\mathit{sn}_s + 1 & \text{if } s.\mathit{ack}_c = s.\mathit{sn}_s + 1 \\ & \leq & u.\mathit{sn}_s & \text{if } s.\mathit{ack}_c \leq s.\mathit{sn}_s. \end{array}$$

17. $$\begin{array}{llll} u.\mathit{ack}_s & = & s.\mathit{ack}_s & \text{if } s.\mathit{ack}_s = \mathtt{nil} \\ & \text{is} & \text{undefined} & \text{if } s.\mathit{mode}_s = \mathtt{closed} \\ & \leq & u.\mathit{sn}_c + 1 & \text{if } s.\mathit{mode}_c = \mathtt{closed} \wedge s.\mathit{ack}_s \neq \mathtt{nil} \\ & = & u.\mathit{sn}_c + 1 & \text{if } s.\mathit{ack}_s = s.\mathit{sn}_c{+}1 \wedge (s.\mathit{mode}_c \neq \mathtt{syn\text{-}sent} \vee (s.\mathit{mode}_c = \mathtt{syn\text{-}sent} \wedge s.\mathit{mode}_s = \mathtt{syn\text{-}rcvd})) \\ & \leq & u.\mathit{sn}_c & \text{if } s.\mathit{ack}_s \leq s.\mathit{sn}_c \vee (s.\mathit{mode}_c = \mathtt{syn\text{-}sent} \wedge \neg(s.\mathit{mode}_s = \mathtt{syn\text{-}rcvd} \wedge \mathit{wait\text{-}t\_o}_s \leq \mathit{now})) \end{array}$$

18. Segment $p \in u.\mathit{in\text{-}transit}_{cs} = p'$ for non-RST segment $(p', t) \in s.\mathit{in\text{-}transit}_{cs}$ but, with:

$$\begin{array}{llll} \mathit{sn}(p) & \leq & u.\mathit{sn}_c & \text{if } s.\mathit{mode}_c = \mathtt{closed} \vee \mathit{sn}(p') \notin \{s.\mathit{ack}_s, s.\mathit{ack}_s + 1\} \\ & = & u.\mathit{sn}_c & \text{if } s.\mathit{sn}_c = \mathit{sn}(p') \wedge (t - \mu) \geq s.\mathit{con\text{-}strt\text{-}time}_c. \\ & < & u.\mathit{sn}_c & \text{if } s.\mathit{mode}_c \neq \mathtt{closed} \wedge (\mathit{sn}(p') < s.\mathit{sn}_c \vee (t - \mu) < s.\mathit{con\text{-}strt\text{-}time}_c). \\ & = & u.\mathit{ack}_s & \text{if } \mathit{sn}(p') = s.\mathit{ack}_s \wedge (p', t) \text{ is a non-SYN segment and } (s.\mathit{mode}_s \neq \mathtt{syn\text{-}rcvd} \vee (s.\mathit{mode}_s = \mathtt{syn\text{-}rcvd} \wedge \mathit{ack}(p') = s.\mathit{sn}_s + 1)). \\ & = & u.\mathit{ack}_s + 1 & \text{if } \mathit{sn}(p') = s.\mathit{ack}_s{+}1 \wedge (p', t) \text{ is a non-SYN segment and } (s.\mathit{mode}_s \neq \mathtt{syn\text{-}rcvd} \vee (s.\mathit{mode}_s = \mathtt{syn\text{-}rcvd} \wedge \mathit{ack}(p') = s.\mathit{sn}_s + 1)). \\ \mathit{ack}(p) & = & u.\mathit{ack}_c & \text{if } \mathit{ack}(p') = \mathit{ack}_c \\ & \leq & u.\mathit{sn}_s + 1 & \text{if } s.\mathit{mode}_c \in \{\mathtt{closed}, \mathtt{syn\text{-}sent}\} \wedge s.\mathit{mode}_s \in \{\mathtt{closed}, \mathtt{listen}\} \vee \mathit{ack}(p') \neq s.\mathit{ack}_c. \\ & = & u.\mathit{sn}_s + 1 & \text{if } \mathit{ack}(p') = s.\mathit{sn}_s + 1 \\ & \leq & u.\mathit{sn}_s & \text{if } \mathit{ack}(p') \leq s.\mathit{sn}_s. \end{array}$$

19. Segment $p \in u.\mathit{in\text{-}transit}_{sc} = p'$ for non-RST segment $(p', t) \in s.\mathit{in\text{-}transit}_{sc}$, but with:

$$
\begin{array}{llll}
sn(p) & \leq & u.sn_s & \text{if } s.mode_s \in \{\texttt{closed}, \texttt{listen}\} \ \vee \ sn(p') \notin \{s.ack_c, s.ack_c + 1\}. \\
 & = & u.sn_s & \text{if } s.sn_s = sn(p') \ \wedge \ (t - \mu) \geq s.con\text{-}strt\text{-}time_s. \\
 & < & u.sn_s & \text{if } s.mode_s \notin \{\texttt{closed}, \texttt{listen}\} \wedge (sn(p') < s.sn_s \ \vee \ (t - \mu) < \\
 & & & con\text{-}strt\text{-}time_s). \\
 & = & u.ack_c & \text{if } sn(p') = s.ack_c \wedge \ (p', t) \text{ is a non-SYN segment.} \\
 & = & u.ack_c + 1 & \text{if } sn(p') = s.ack_c + 1 \wedge \ (p', t) \text{ is a non-SYN segment.} \\
ack(p) & = & u.ack_s & \text{if } ack(p') = ack_s \ \wedge \ (t - \mu) \geq s.con\text{-}strt\text{-}time_s \\
 & \leq & u.sn_c + 1 & \text{if } s.mode_s \in \{\texttt{closed}, \texttt{listen}\} \ \wedge \ s.mode_c = \texttt{closed} \ \vee ack(p') \neq \\
 & & & s.ack_s. \\
 & = & u.sn_c + 1 & \text{if } ack(p') = s.sn_c + 1 \ \wedge \ (p', t) \text{ is a SYN segment.} \\
 & = & u.sn_c + 1 & \text{if } ack(p') = s.sn_c + 1 \wedge s.mode_c \neq \texttt{syn-sent} \wedge (p', t) \text{ is a non-SYN} \\
 & & & \text{segment.} \\
 & \leq & u.sn_c & \text{if } ack(p') \leq s.sn_c \ \wedge \ (p', t) \text{ is a SYN segment.} \\
 & \leq & u.sn_c & \text{if } ack(p') \leq s.sn_c \ \vee s.mode_c = \texttt{syn-sent} \ \wedge \ (p', t) \text{ is a non-SYN} \\
 & & & \text{segment.}
\end{array}
$$

The simulation of steps

In this section we proof that F_{BN} is indeed a timed forward simulation from the states of $\mathcal{BTCP}^h$ to the states of $\mathcal{NTCP}$ by showing the correspondence of actions.

Lemma 8.2

$\mathcal{BTCP}^h \leq_F^t \mathcal{NTCP}$ via F_{BN}.

Proof: We prove that F_{BN} is a timed refinement mapping from $\mathcal{BTCP}^h$ to $\mathcal{NTCP}$ with respect to I_T and I_B by showing that the two cases of Definition 3.12 are satisfied.

Base Case

In the start state s_0 of $\mathcal{BTCP}^h$ we have $s_0.mode_c = \texttt{closed}$, $s_0.mode_s = \texttt{closed}$, $s_0.now = 0$, $s_0.in\text{-}transit_{cs} = \emptyset$, $s_0.in\text{-}transit_{sc} = \emptyset$, and and all other variables undefined. The start state, u_0 of $\mathcal{NTCP}$ has $u_0.mode_c = \texttt{closed}$, $u_0.mode_s = \texttt{closed}$, $u_0.now = 0$ $u_0.sn_c = 0$, $u_0.sn_s = 0$, $u_0.in\text{-}transit_{cs} = \emptyset$, $u_0.in\text{-}transit_{sc} = \emptyset$, and all other variables undefined. Therefore, it is clear that $F_{\text{BN}}(s_0) \cap u_0 \neq \emptyset$.

Inductive Case

Assume $(s, a, s') \in Steps(\mathcal{BTCP}^h)$. Below we consider cases based on a and for each case we define a finite execution fragment α of S such that $fstate(\alpha) = F_{\text{BN}}(s)$, $lstate(\alpha) = F_{\text{BN}}(s')$, and $t\text{-}trace(\alpha) = t\text{-}trace(s, a, s')$. For the steps of the proof below we do not include the time of occurrence and last time in the *timed traces* of (s, a, s') or α, so as not to clutter the proof. However, it is clear that since the time-passage steps in $\mathcal{NTCP}$ have no restrictions, if we show $trace(\alpha) = trace(s, a, s')$ then $t\text{-}trace(\alpha) = t\text{-}trace(s, a, s')$. We use u and u' to denote $F_{\mathrm{BN}}(s)$ and $F_{\mathrm{BN}}(s')$ respectively. If an action has a symmetric counterpart from the other host we will not show the proof of correspondence for that action.

<u>*a = send-msg*$_c$*(open, m, close)*.</u>

For this case, $\alpha = (u, a, u')$. There are two interesting cases for this step. The first is if $s.mode_c$ = `syn-sent` $\wedge$ $s.send\text{-}buf_c = \epsilon$ and *close* is true. The second interesting case for this step is when $s.mode_c$ = `closed` and *open* is true. For all other states, since the steps only affect variables that are equal to each other in the relation, or have no effect, it is easy to see that after a the resulting state, u', is in $F_{\mathrm{BN}}(s')$.

1. The case where $s.mode_c$ = `syn-sent` $\wedge\, s.send\text{-}buf_c = \epsilon$ and *close* is true is interesting because after the step $s'.mode_c$ = `closed`, so the set of allowable values for $u'.ack_s$, $u'.in\text{-}transit_{cs}$, and $u'.in\text{-}transit_{sc}$, may change. For $u'.ack_s$ the relation is affected if $s.ack_s = s.sn_c + 1$. In the corresponding set of states $u.ack_s = u.sn_c + 1$. After the steps $u'.ack_s$ should be less than or equal to $u'.sn_c + 1$. This is clearly true. For segments $p \in u'.in\text{-}transit_{cs}$, after α, $sn(p)$ should be less than or equal to $u'.sn_c$. Since $s.mode_c$ = `syn-sent` and Invariant 8.1 tells us that all segments (p, t) with $t - \mu \geq s.con\text{-}strt\text{-}time_c$ have $sn(p) = s.sn_c$, by F_{BN}, we know that all segments $p \in u.in\text{-}transit_{cs}$, $sn(p) \leq u.sn_c$. Since the step does not change $u.in\text{-}transit_{cs}$ or $u.sn_c$, we have $sn(p) \leq u'.sn_c$ for $p \in u'.in\text{-}transit_{cs}$. For a segment $q \in u.in\text{-}transit_{sc}$, the relation is affected if the corresponding segment $(q', t) \in s.in\text{-}transit_{sc}$ has $ack(q') = s.sn_c + 1$, or $ack(q') \leq s.sn_c$. After step (s, a, s'), F_{BN}, says that all segments $q \in u'.in\text{-}transit_{sc}$ must have $ack(q) \leq u'.sn_c + 1$. Since, in the states corresponding to state s, we know that $ack(q) \leq u.sn_c + 1$, we clearly have the right correspondence for this variable.

2. For the case where $s.mode_c$ = `closed` and *open* is true, the difficulty in showing u' is in the set of states defined by F_{BN}, lies in showing that $u'.ack_s$, $u'.in\text{-}transit_{cs}$, and $u'.in\text{-}transit_{sc}$ have values that are in the set of values defined by F_{BN}. The criterion used by F_{BN} for defining the set of allowable values for $u'.ack_s$ may change after step (s, a, s') if $s'.mode_s \neq$ `closed` $\wedge$ $s'.ack_s \neq$ `nil`. Since $s.ack_s$ does not change in $\mathcal{BTCP}^h$ after (s, a, s'), nor does $u.ack_s$ change after α in $\mathcal{NTCP}$, we need to show

that even if the criterion changes, $u.ack_s$ is in the set of allowable values for $u'.ack_s$ as defined by F_{BN}. The change in criterion comes about because $s.mode_c$ = `closed` while $s'.mode_c$ = `syn-sent`. By F_{BN}, $u.ack_s \leq u.sn_c + 1$. From Invariant 8.5 we know that if $s'.mode_s$ = `syn-rcvd` then $s'.ack_s \leq s'.sn_c$. Therefore, by F_{BN}, $u'.ack_s$ should be less than or equal to $u'.sn_c$ which we know to be true because $u'.sn_c = u.sn_c + 1$.

The criterion used by F_{BN} for defining the set of allowable values for the sequence numbers of segments $p \in u.in\text{-}transit_{cs}$ changes after step (s, a, s') if $s.mode_s \in$ {`closed`, `listen`}. Because the sequence numbers do not change after the step, we need to show that the values are in the new set of allowable values. If $s.mode_c$ = `closed` and $s.mode_s \in$ {`closed`, `listen`}, then by F_{BN} for all segments p, $sn(p) \leq u.sn_c$. Since the timestamp minus μ of every segment in $s.in\text{-}transit_{cs}$, is less than $s'.con\text{-}strt\text{-}time_c$ (Invariant 8.2), by F_{BN}, for all segments p, $sn(p) < u'.sn_c$, which is clearly true.

For segments $p \in u.in\text{-}transit_{sc}$, the criterion used for determining the allowable values for $ack(p)$, changes after step (s, a, s') if $s.mode_s \in$ {`closed`, `listen`}. Because the acknowledgment numbers do not change after the step, we need to show that the values are in the new set of allowable values. If $s.mode_c$ = `closed` and $s.mode_s \in$ {`closed`, `listen`}, then by F_{BN} for all segments p, $ack(p) \leq u.sn_c + 1$. After the step, we know by Invariant 8.6, that for all SYN segments $(p', t) \in s'.in\text{-}transit_{sc}$, $ack(p') \leq s'.sn_c$. After step (u, a, u'), since $u'.sn_c = u.sn_c + 1$, we know that $ack(p) \leq u'.sn_c$, which is in the set of allowable values defined by F_{BN}. For non-SYN segments, since $s'.mode_c$ = `syn-sent`, we again have by F_{BN} that $ack(p)$ should less than or equal to $u'.sn_c$, which we know is true.

a = *passive-open*.

For this step the corresponding $\alpha = (u, a, u')$. It is easy to see that $u' \in F_{\text{BN}}(s')$, because the step does not change any variables that are not equal in the relation.

a = *send-msg$_s$(m, close)*.

For this step the corresponding $\alpha = (u, a, u')$. This is another step that only changes variables that are equal in the relation, so it is easy to see that $u' \in F_{\text{BN}}(s')$.

a = *send-seg*$_{cs}$(*SYN*, sn_c).

For this step, the corresponding $\alpha = (u, a, u')$. Step (s, a, s') adds a SYN segment (p', t) to $s.in\text{-}transit_{cs}$ with $sn(p') = s.sn_c$, and α adds a SYN segment p to $u.in\text{-}transit_{cs}$ with $sn(p) = u.sn_c$. Clearly $u'.in\text{-}transit_{cs}$ is in the set of allowable states as defined by F_{BN}.

a = *receive-seg*$_{cs}$(*SYN*, sn_c).

We break the proof of correspondence for this step into two cases.

1. The first case is if $s.mode_s$ = `closed`. For this case the corresponding α is the empty step. In Chapter 3 we defined λ to be the empty action, so we have $\alpha = (u, \lambda, u')$. For this case, after step (s, a, s'), $s'.send\text{-}rst_s$ = `true`, $s.ack_s = [sn_c] + 1$, and $s'.rst\text{-}seq_s = 0$. However, since $s'.mode_s$ = `closed`. The empty step gives the right set of corresponding states.

2. The second case is for all other states, s. For this case $\alpha = (u, a, u')$. Let (p', t') be the SYN segment received by (s, a, s') and p be the SYN segment received by α. This step is interesting if $s.mode_s$ = `listen`. For most variables it is clear that their values in state u' are in the set of allowable values defined by F_{BN}. The interesting cases are for $u'.ack_s$, $u'.sn_c$, $u'.ack_c$, $u'.in\text{-}transit_{sc}$, and $u'.in\text{-}transit_{cs}$. We know $u.ack_s$ = `nil` by F_{BN}. After a, in $\mathcal{BTCP}^h$, if $s'.mode_c \neq$ `closed` and $s'.ack_s = s'.sn_c + 1$ or $s'.ack_s \leq s'.sn_c$, then $sn(p') = s'.sn_c$ or $sn(p') < s'.sn_c$. We know from Invariant 8.3 that if $sn(p') = s'.sn_c$ then $t - \mu \geq s'.con\text{-}strt\text{-}time_c$, so in the corresponding set of states $sn(p) = u'.sn_c$. If $sn(p') < s'.sn_c$, then we know from Invariant 8.1 that $t - \mu < s'.con\text{-}strt\text{-}time_c$, so in the corresponding set of states $sn(p) < u'.sn_c$, so we have the correct value for this case. From Invariant 8.11 we know that if $s'.mode_c \neq$ `closed` then $s'.ack_s \leq s'.sn_c + 1$. Therefore, the only other possibility is if $s'.mode_c$ = `closed`, then $u'.ack_s \leq u'.sn_c + 1$, which by Invariant 7.2 we know is true.

   If $s'.ack_c = s'.sn_s + 1$, then the criterion used by F_{BN} for defining the set of allowable values for $u'.sn_c$ changes if $s'.sn_c = s'.ack_s$ or $s'.sn_c = s'.ack_s + 1$. However, since $s'.mode_s$ = `syn-rcvd`, Invariant 8.7 tells us that if $s'.ack_c \neq$ `nil`, then $s'.ack_c \leq s'.sn_s$. Therefore, the criterion for defining the allowable values for this variable does not change. For $u'.ack_c$, we already know that if $s'.ack_c \neq$ `nil`, then it is less than or equal to $s'.sn_c$. Since $s.mode_s = \texttt{listen}$ by $F_{\rm BN}$, $u.ack_c \leq u.sn_s + 1$. By $F_{\rm BN}$ $u'.ack_c$ should be less than or equal to $u'.sn_s$. Since $u.ack_c \leq u.sn_s+1$, and $u'.sn_s = u.sn_s+1$, $u'.ack_c$ is indeed less than or equal to $u'.sn_s$.

The criterion used by $F_{\rm BN}$ for defining the set of allowable values for the sequence numbers and acknowledgment numbers of segments $q \in u.in\text{-}transit_{sc}$ changes after step (s, a, s') if $s.mode_c \in \{\texttt{closed}, \texttt{syn-sent}\}$. Because the numbers do not change after the step, we need to show that the values are in the new set of allowable values. We first examine the sequence numbers. If $s.mode_s = \texttt{listen}$ and $s.mode_c \in \{\texttt{closed}, \texttt{syn-sent}\}$, then by $F_{\rm BN}$ for all segments q, $sn(q) \leq u.sn_s$. Since the timestamp minus μ on every segment must be less than $s'.con\text{-}strt\text{-}time_s$ (Invariant 8.2), by $F_{\rm BN}$, for all segments q, $sn(q) < u'.sn_s$, which is clearly true. Also because the timestamp minus μ on every segment must be less than $s'.con\text{-}strt\text{-}time_s$, the allowable values for acknowledgment numbers of segments on $u'.in\text{-}transit_{sc}$ is not affected.

The criterion used by $F_{\rm BN}$ for defining the set of allowable values for the acknowledgment numbers of segments $q \in u.in\text{-}transit_{cs}$ also changes after this step if $s.mode_c \in \{\texttt{closed}, \texttt{syn-sent}\}$. It does not change for sequence numbers because we know from Invariant 8.8, that for all segments $q' \in s'.in\text{-}transit_{cs}$, $ack(q) \leq s'.sn_s$. For acknowledgment numbers, since $s.mode_s = \texttt{listen}$, if $s.mode_s \in \{\texttt{closed}, \texttt{syn-sent}\}$, then by $F_{\rm BN}$ for all SYN segments q, $ack(q) \leq u.sn_s + 1$. After the step, we know by Invariant 8.8, that for all segments $q' \in s'.in\text{-}transit_{cs}$, $ack(q) \leq s'.sn_s$. After step (u, a, u'), since $u'.sn_s = u.sn_s + 1$, we know that $ack(q) \leq u'.sn_s$, which is in the set of allowable values defined by $F_{\rm BN}$.

<u>$a = send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$.</u>

For this step the corresponding $\alpha = (u, a, u')$. Let (p', t') be the SYN segment added by (s, a, s') and p be the SYN segment added by α. Thus, we have $sn(p) = u'.sn_s$ and $ack(p) = u'.ack_s$; and $sn(p') = s'.sn_s$ and $ack(p) = s'.ack_s$. Since $s.mode_s = \texttt{syn-rcvd}$ when this step is enabled, the sequence number of p is right as defined by $F_{\rm BN}$. If in $\mathcal{BTCP}^h$, $ack(p') = s'.sn_c + 1$ or $ack(p') \leq s'.sn_c$, then by $F_{\rm BN}$, $ack(p)$ should be equal to

$u'.sn_c + 1$ or $ack(p)$ should be less than $u'.sn_c$ respectively. Since if $s.ack_s = s.sn_c + 1$ then $u.ack_s = u.sn_c + 1$ or if $s.ack_s \leq s.sn_c$ then $u.ack_s \leq u.sn_c$, we clearly have the right set of values.

$\underline{a = \textit{receive-seg}_{sc}(SYN,\ sn_c,\ ack_s)}$.

For this step we have two cases.

1. The first case is if $s.mode_c = \texttt{closed} \vee s.mode_c = \texttt{syn-sent} \wedge ack(p') \neq s.sn_c + 1$. For this case the corresponding α is the empty step. Since for this case (s, a, s') only changes $s.\textit{send-rst}_c$ and $s.\textit{rst-seq}_c$, it is clear that the empty step gives the right set of corresponding states.

2. The second case is for all other states. For this case $\alpha = (u, a, u')$. Let (p', t) be the segment received in step (s, a, s'), and p be the segment received in step α. If $s.mode_c = \texttt{syn-sent} \wedge ack(p') = s.sn_c + 1$, then after step (s, a, s'), $s'.mode_c = \texttt{estb}$ and $s'.ack_c = sn(p')+1$. In the corresponding state of $\mathcal{NTCP}$, $u.mode_c = \texttt{syn-sent} \wedge ack(p) = u.sn_c + 1$, and after α, $u'.mode_c = \texttt{estb}$ and $u'.ack_c = sn(p) + 1$. We need to show that this value of $u'.ack_c$ is in the set of acceptable values as defined by F_{BN}. If $s'.mode_s \in \{\texttt{closed}, \texttt{listen}\}$ then $u'.ack_c \leq u'.sn_s + 1$ which we know to be true by Invariant 7.2. If $s'.mode_s \notin \{\texttt{closed}, \texttt{listen}\}$ then the allowed values for $u'.ack_c$ depend on the relationship of $sn(p')$ to $s'.sn_s$. If $s'.ack_c = s'.sn_s+1$ or $s'.ack_c \leq s'.sn_c$, then by F_{BN}, $u'.ack_c$ should have the same relationship to $u'.sn_s$. If $s'.ack_c = s'.sn_s+1$ or $s'.ack_c \leq s'.sn_c$ then $sn(p') = s'.sn_s$ or $sn(p') \leq s'.sn_s$ respectively. We know from Invariant 8.3 that if $sn(p') = s'.sn_s$ then $t - \mu \geq s'.\textit{con-strt-time}_s$, so in the corresponding set of states $sn(p) = u'.sn_s$. If $sn(p') < s'.sn_s$, then we know from Invariant 8.1 that $t - \mu < s'.\textit{con-strt-time}_s$, so in the corresponding set of states $sn(p) < u'.sn_s$, so we have the correct set if values for this case also.

   The change of $s.ack_c$ from $\texttt{nil}$ to $sn(p')+1$, may also change the criterion used by F_{BN} for defining the set of allowable values for $u'.sn_s$, if $s.mode_s \notin \{\texttt{closed}, \texttt{listen}, \texttt{syn-rcvd}\}$. However, from Invariant 8.12 we know that if $s'.sn_s \in \{ack_c, ack_c+1\}$ then $s'.mode_s = \texttt{syn-rcvd}$, so the criterion does not changes. This change may also change the set of allowable values for acknowledgment numbers of segments in $u'.\textit{in-transit}_{cs}$, and sequence numbers of non-SYN segments in $u'.in\text{-}transit_{sc}$. For acknowledgment numbers of segments in $s'.in\text{-}transit_{cs}$, the change may have an effect for any segment $q \in u'.in\text{-}transit_{cs}$ that has corresponing segment (q', t') such that $ack(p') \neq s'.sn_s$. However, we know by Invariant 8.9 that $sn(p') \geq ack(q')$ for any segment $(q', t') \in s.in\text{-}transit_{cs}$. Therefore, we know that $s'.ack_c > ack(q')$. From Invariant 7.3 we know that for any corresponding segments $q \in u'.in\text{-}transit_{cs}$, $ack(q) \leq u'.sn_s + 1$, so we have the correct correspondence of states. If $s.mode_s \in \{\texttt{closed}, \texttt{listen}\}$, the change of $s.ack_c$ from $\texttt{nil}$ to $sn(p) + 1$ may change the allowable set of values for sequence numbers for non-SYN segments in $u'.in\text{-}transit_{sc}$. However, from Invariant 8.10 we know that for all non-SYN segments $(q', t') \in s'.in\text{-}transit_{sc}$, $sn(q') \notin \{s'.ack_c, s'.ack_c + 1\}$, and from Invariant 7.1 we know $sn(p) \leq u'.sn_c$, so we have the correct correspondence of states.

$\underline{a = prepare\text{-}msg_c}$.

For this step $\alpha = (u, a, u')$. It is easy to see that for most variables the relationship is maintained after α. The difficulty lies in showing that the relationship is preserved for $u'.sn_c$, $u'.ack_s$, $u'.in\text{-}transit_{cs}$, and $u'.in\text{-}transit_{sc}$. We first examine the case for $u'.sn_c$. By Invariant 8.14 we know that $s.sn_c + 1 \geq s.ack_s$, if $s.ack_s \neq \texttt{nil} \wedge s.mode_s \neq \texttt{closed}$. If after this step $s'.sn_c = s'.ack_s$, or $s'.sn_c = s'.ack_s + 1$ (if sn_c is incremented twice), then it must be that $s.ack_s = s.sn_c + 1$. By F_{BN}, in the related set of states $u.ack_s = u.sn_c + 1$, so after α, $u'.sn_c = u'.ack_s$ or $u'.sn_c = u'.ack_s + 1$ respectively. If $s.sn_c \geq s.ack_s \ \vee s.ack_s = \texttt{nil} \ \vee s.mode_s = \texttt{closed}$ then after (s, a, s'), $s'.sn_c = s'.ack_s + 1$ or $s'.sn_c \notin \{s'.ack_s, s'.ack_s + 1\}$. After α we know $u'.sn_c = u'.ack_s + 1$, and/or $u'.sn_c > \max(u'.max\text{-}u\text{-}sn_c, u'.ack_s - 1)$, which is the set of allowable values for this state.

For the case of $u'.ack_s$, we know that if $s.ack_s \neq \texttt{nil} \wedge s.mode_s \neq \texttt{closed}$ then $s.sn_c + 1 \geq s.ack_s$ (Invariant 8.14). Therefore, after this step $s'.ack_s \leq s'.sn_c$. In the corresponding set of states we know $u.ack_s \leq u.sn_c + 1$, so after α, $u'.ack_s \leq u'.sn_c$, which gives the correct correspondence of states.

For sequence numbers of segments $p \in u'.in\text{-}transit_{cs}$, the set of allowable values may change if after (s, a, s') there is a segment $(p', t) \in s'.in\text{-}transit_{cs}$ with $sn(p') = s'.sn_c$. However, from Invariant 8.13 we know that this is not the case. By Invariant 8.15 we know that $ack(p') \leq s.sn_c + 1$, for segments $(p', t) \in s.in\text{-}transit_{sc}$. Therefore in the corresponding set of states, for segments $p \in u.in\text{-}transit_{sc}$, $ack(p) \leq u.sn_c + 1$. Therefore, after this step the correspondence is maintained.

$\underline{a = prepare\text{-}msg_s}$.

This step is symmetric to $a = prepare\text{-}msg_c$,

$\underline{a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)}$.

For this step $\alpha = (u, a, u')$. Let (p', t') be the segment added by (s, a, s') and p be the segment added by α. Thus, we have $sn(p) = u'.sn_c$ and $ack(p) = u'.ack_c$; and $sn(p') = s'.sn_c$ and $ack(p) = s'.ack_c$. If in $\mathcal{BTCP}^h$ $sn(p') = s'.ack_s$ or $sn(p') = s'.ack_s + 1$, then by F_{BN}, $sn(p)$ should be equal to $u'.ack_s$ or $sn(p)$ should be equal to $u'.ack_s + 1$ respectively. Since if $s.sn_c = s.ack_s$ then $u.sn_c = u.ack_s$, or if $s.sn_c = s.ack_s + 1$ then $u.sn_c = u.ack_s + 1$, we clearly get the correct corresponding states. Also, if in $\mathcal{BTCP}^h$ $ack(p') = s'.sn_s + 1$ or $ack(p') \leq s'.sn_s$, then by F_{BN}, $ack(p)$ should be equal to $u'.sn_s + 1$ or $ack(p)$ should be less than $u'.sn_s$ respectively. Since if $s.ack_c = s.sn_s + 1$ then $u.ack_c = u.sn_s + 1$ or if $s.ack_c \leq s.sn_s$ then $u.ack_c \leq u.sn_s$, we clearly have the right set of values.

$\underline{a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)}$.

We break the proof of correspondence for this step into the usual two cases.

1. The first case is if $(s.mode_s \in \{\texttt{closed}, \texttt{listen}\}) \vee (s.mode_s = \texttt{syn-rcvd} \wedge ack(p') \neq s.sn_s + 1)$. For this case the corresponding α is the empty step. It is clear that the empty step gives the right set of corresponding states.

2. The second case is for all other states. For this case $\alpha = (u, a, u')$. Let (p', t) be the segment received in step (s, a, s') and p be the segment received by in step α. We break the second case into several subcases. The subcases are not necessarily disjoint set of states.

   (a) The first subcase occurs if $sn(p') = s.ack_s$. After a, in $\mathcal{BTCP}^h$, $s'.ack_s = sn(p') + 1$. In the corresponding set of states $sn(p) = u.ack_s$, and after α $u'.ack_s = sn(p) + 1$. We need to show that $u'.ack_s$, $u'.sn_c$, $u'.in\text{-}transit_{cs}$, and $u'.in\text{-}transit_{sc}$ all have allowable values after α. We first look at the case for $u'.ack_s$. If $s'.ack_s =$

$s'.sn_c{+}1$ then $sn(p') = s.sn_c$. By Invariant 8.4 we know that if $sn(p') = s.ack_s$ and $sn(p') = s.sn_c$, then $t - \mu \geq s'.con\text{-}strt\text{-}time_c$, so in the corresponding set of states $sn(p) = u.sn_c$. If $s'.ack_s \leq s'.sn_c$ then $sn(p') < s.sn_c$, so in the corresponding set of states $sn(p) < u.sn_c$. Thus, the correspondence holds in either of these cases. If $s.mode_c = \texttt{closed}$, the $u'.ack_s$ should be less than or equal to $u'.sn_c + 1$. From Invariant 7.2 we know this is true.

If $s.mode_c \neq \texttt{closed}$, then the criterion for determining allowable values for $u'.sn_c$ may change. However, we know from Invariant 8.17 that if $s'.mode_c \in$ *sync-states*, then $sn(p') = s'.sn_c$. Therefore, after step (s, a, s') either $s'.mode_c \in \{\texttt{closed}, \texttt{syn-sent}\}$ or $s'.sn_c \notin \{s'.ack_s, s'.ack_s + 1\}$. Therefore, the criterion for the allowable values for this variable does not change.

For non-SYN segments in $u'.in\text{-}transit_{cs}$ the criterion for determining allowable values for sequence numbers may change if there is a non-SYN segment $(q', t') \in s'.in\text{-}transit_{cs}$ with $sn(q') \in \{s'.ack_s, s'.ack_s + 1\}$. However, from Invariant 8.18 we know that for all non-SYN segments $(q', t') \in s'.in\text{-}transit_{cs}$, $sn(q') \notin \{s'.ack_s, s'.ack_s + 1\}$, and from Invariant 7.1 we know that $sn(q) \leq u'.sn_c$ for segment $q \in u'.in\text{-}transit_{cs}$. For $u'.in\text{-}transit_{sc}$, if there exists a segment $(q', t') \in s'.in\text{-}transit_{sc}$ with $ack(q') = s'.ack_s$, then the corresponding segment $q \in u'.in\text{-}transit_{sc}$ should have $ack(q) = u'.ack_s$. However, from Invariant 8.16 we know there are no such segments $(q', t') \in s'.in\text{-}transit_{sc}$.

(b) The second subcase occurs if $ack(p') = s.sn_s + 1$ and $s.mode_s = \texttt{last-ack}$. For this case, after step (s, a, s'), $s'.mode_s = \texttt{closed}$, and after α, $u'.mode_s = \texttt{closed}$. This change may affect the states of $\mathcal{NTCP}$ related to $s'.sn_s$, $s'.sn_c$, $s'.ack_c$, $s'.in\text{-}transit_{sc}$, and $s'.in\text{-}transit_{cs}$.

We first look at the case for $u'.sn_s$. After α, $u'.sn_s$ should be greater than or equal to $\max(u'.max\text{-}u\text{-}sn_s, u.ack_c - 1)$. From Invariants 7.1, and 7.3 we know this is true. If $s.sn_c = s.ack_s$ or $s.sn_c = s.ack_s + 1$ the criterion for the allowable values for $u.sn_c$ changes after α, because $s'.ack_s$ is undefined. However, again from Invariants 7.1, and 7.3 we know $u'.sn_c \geq \max(u'.max\text{-}u\text{-}sn_c, u'.ack_s - 1)$, which is in the set of correct corresponding states. After, α, it is also clear that

$u'.ack_c \leq u'.sn_s + 1$, which by is the correct set of allowable values for $u'.ack_c$.

The criterion used by F_{BN} for defining the set of allowable values for the sequence numbers and acknowledgment numbers of any segment $q \in u.\mathit{in\text{-}transit}_{sc}$ changes after this step if $s.mode_c \in \{\mathtt{closed}, \mathtt{syn\text{-}sent}\}$. After this step, by F_{BN}, any $q \in u'.\mathit{in\text{-}transit}_{sc}$, should have $sn(q) \leq u'.sn_s$, and $ack(q) \leq u'.sn_c + 1$. By Invariants 7.1 and 7.3 we know that this is true.

The criterion used by F_{BN} for defining the set of allowable values for the sequence numbers and acknowledgment numbers of segments $q \in u.\mathit{in\text{-}transit}_{cs}$ also changes after this step if $s.mode_c \in \{\mathtt{closed}, \mathtt{syn\text{-}sent}\}$. After this step, by F_{BN}, any $q \in u'.\mathit{in\text{-}transit}_{cs}$, should have $sn(q) \leq u'.sn_c$, and $ack(q) \leq u'.sn_s + 1$, again by Invariants 7.1 and 7.3 we know that this is true.

(c) The fourth subcase is for all other states that does not involve any of the changes of the previous subcases. In these states, if any changes are made it only involves variables that are equal to each other in the relation, so the correspondence of states is preserved after (s, a, s') and α.

$\underline{a = \mathit{send\text{-}seg}_{sc}(sn_s,\ ack_s,\ msg_s)}$.

This is symmetric to $a = \mathit{send\text{-}seg}_{cs}(sn_c,\ ack_c,\ msg_c)$.

$\underline{a = \mathit{receive\text{-}seg}_{sc}(sn_s,\ ack_s,\ msg_s)}$.

For this step $\alpha = (u, a, u')$. This step is symmetric to $a = \mathit{receive\text{-}seg}_{cs}(sn_c,\ ack_c,\ msg_c)$, except that there is no subcase symmetric to subcase 2(a) of that step.

$\underline{a = \mathit{receive\text{-}msg}_c(m)}$.

For this step the corresponding $\alpha = (u, a, u')$. It is easy to see that $u' \in F_{\mathrm{BN}}(s')$.

$\underline{a = \mathit{receive\text{-}msg}_s(m)}$.

For this step the corresponding $\alpha = (u, a, u')$. It is easy to see that $u' \in F_{\mathrm{BN}}(s')$.

$\underline{a = \mathit{send\text{-}seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$.

For this step $\alpha = (u, a, u')$. The proof of correspondence for this step is the same as the proof for $a = \mathit{send\text{-}seg}_{cs}(sn_c,\ ack_c,\ msg_c)$.

$\underline{a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$.

The proof of correspondence for this step is essentially the same as the proof for the step with $a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c)$. The only exception is that there is no case 2(b) for this step.

$\underline{a = \textit{send-seg}_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)}$.

For this step $\alpha = (u, a, u')$. This case is symmetric to the case for $a = \textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$.

$\underline{a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)}$

For this step $\alpha = (u, a, u')$. This case is symmetric to the case for $a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$.

$\underline{a = \textit{time-out}_s}$.

We break the proof of correspondence for this step into two cases.

1. The first case occurs when $s.mode_c = \texttt{timed-wait} \wedge now \geq \textit{first}(\textit{t-out}_c)$. For this case the $\alpha = (u, a, u')$. After step (s, a, s'), $s'.mode_s = \texttt{closed}$, and after α, $u'.mode_s = \texttt{closed}$. The proof that u' is in the set of states that are related to s', is the same as the proof for subcase 2(b) for the step with $a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c)$.

2. The second case is for all other states of $\mathcal{BTCP}^h$. For these states $\alpha = (u,\ \textit{set-reset}_s,\ u')$. It is easy to see that $u' \in F_{\text{BN}}(s')$, for this case.

$\underline{a = \textit{timeout}_c}$.

This step is symmetric to $a = \textit{timeout}_s$.

$\underline{a = \textit{crash}_c}$.

For this step $\alpha = (u, a, u')$. It is easy to see that $u' \in F_{\text{BN}}(s')$.

$\underline{a = \textit{crash}_s}$.

This step is symmetric to $a = \textit{crash}_c$.

$\underline{a = \textit{recover}_s}$.

For this step $\alpha = (u, a, u')$. After this step, $s'.mode_s = \texttt{closed}$. The proof that u' is in the set of states that are related to s', is the same as for subcase 2(b) for the step with $a = \textit{receive-seg}_{cs}(sn_s,\ ack_s,\ msg_s)$.

$\underline{a = recover_c.}$

This step is symmetric to $a = recover_s$.

$\underline{a = drop_{cs}(p', t)}$ (from the $\mu Ch_{cs}(\mathcal{P})$ component of $\mathcal{BTCP}^h$).

For this step we have two cases. The first case is if (p', t) is a reset segment. For that case α is the empty step and it is clear that we get the correct corresponding states. The second case is if (p', t) is not a reset segment. For this case α = *(u, $drop_{cs}(p), u'$)*, where p is the segment constructed from (p') by relation F_{BN}. It is easy to see that $u' \in F_{\text{BN}}(s')$.

$\underline{a = drop_{sc}(p't)}$ (from the $\mu Ch_{sc}(\mathcal{P})$ component of $\mathcal{BTCP}^h$) .

This step is symmetric to $a = drop_{cs}(p', t)$.

$\underline{a = duplicate_{cs}(p', t)}$ (from the $\mu Ch_{cs}(\mathcal{P})$ component of $\mathcal{BTCP}^h$).

For this we have two cases. The first case is if (p', t) is a reset segment. For that case α is the empty step and it is clear that we get the correct corresponding states. The second case is if (p', t) is not a reset segment. For this case α = *(u, $duplicate_{cs}(p), u'$)*, where p is the segment constructed from (p') by relation F_{BN}. It is easy to see that $u' \in F_{\text{BN}}(s')$.

$\underline{a = duplicate_{sc}(p', t)}$ (from the $Ch_{sc}(\mathcal{P})$ component of $\mathcal{TCP}^h$).

This step is symmetric to $a = duplicate_{cs}(p', t)$.

$\underline{a = \nu(t) \text{ (time-passage)}}$

The corresponding $\alpha = (u, \nu(t), u')$. It is easy to see that $u' \in F_{\text{BN}}(s')$.

$\underline{a = send\text{-}seg_{sc}(RST,\ ack_s,\ rst\text{-}seq_s)}$.

For this step we have two cases. α is the empty step, and it is easy to see that $u' \in F_{\text{BN}}(s')$.

$\underline{a = receive\text{-}seg_{cs}(RST,\ ack_s,\ rst\text{-}seq_c)}$.

For this step we have two cases.

1. The first case is if it is not a valid reset segement. For that case, α is the empty step, and it is easy to see that $u' \in F_{\text{BN}}(s')$.

2. The second case is if the segment contains a valid reset. For this case α = *(u, $set\text{-}reset_s$, u')*. It is easy to see that $u' \in F_{\text{BN}}(s')$, for this case.

$\underline{a = send\text{-}seg_{cs}(RST,\ ack_c,\ rst\text{-}seq_c)}$.

For this step α is the empty step, and it is easy to see that $u' \in F_{\text{BN}}(s')$.

a = *receive-seg*$_{sc}$*(RST, ack*$_s$*, rst-seq*$_s$*)*.

The proof of correspondence for this step is symmetric to the proof of correspondence for the step with a = *receive-seg*$_{cs}$*(RST, ack*$_s$*, rst-seq*$_c$*)*.

a = *shut-down*$_s$.

For this step $\alpha = (u, a, u')$. After this step, $s'.mode_s$ = `closed`. The proof that u' is in the set of states that are related to s', is the same as for subcase 2(b) for the step with a = *receive-seg*$_{cs}$*(sn*$_s$*, ack*$_s$*, msg*$_s$*)*.

a = *shut-down*$_c$.

This case is symmetric to the case for a = *shut-down*$_s$.

a = *clock-counter-tick*$_c$ and a = *clock-counter-tick*$_s$

For these steps the corresponding step α of $\mathcal{NTCP}$ is (u, λ, u'). Clearly the traces are the same, since *clock-counter-tick*$_c$ and *clock-counter-tick*$_s$ are internal. Since this step of $\mathcal{NTCP}$ does not affect any variables that affect relation $F_{\rm BN}$, it is clear that $u' \in F_{\rm BN}(s')$.

This concludes the simulation proof. ■

8.4.5 Proof of trace inclusion

We can now proof that the GTA model of TCP with bounded counters, $\mathcal{BTCP}$, implements a patient version of Specification S.

Theorem 8.2

$\mathcal{BTCP} \sqsubseteq_t$ *patient*(S).

Proof: From Lemma 8.2 we get that $\mathcal{BTCP}^h \leq_F^t \mathcal{NTCP}$, which because of the soundness of timed forward simulation (Theorem 3.7) and the soundness of adding history variables (Theorem 3.9) implies that $\mathcal{BTCP} \sqsubseteq_t \mathcal{NTCP}$. From Theorem 8.1 we know $\mathcal{NTCP} \sqsubseteq$ *patient*(S). Thus, we now have $\mathcal{BTCP} \sqsubseteq_t \mathcal{NTCP}$ and $\mathcal{NTCP} \sqsubseteq_t$ *patient*(S). Therefore, since the subset relation and thus the implements relation is transitive we get $\mathcal{BTCP} \sqsubseteq_t$ *patient*(S). ■

This concludes our work on TCP. In the next chapter we start the examination of T/TCP.

Chapter 9

T/TCP

In this chapter we present the general timed automaton for T/TCP. The automaton we specify has the TCP accelerated open (TAO) mechanism, but does not include the features to truncate timed-wait state. We do not include the features to truncate timed-wait state, because, first the TAO mechanism is the more important and the more interesting of the two features incorporated into T/TCP, and second, even with just this one new feature the protocol is sufficiently different from TCP to make its verification interesting. We also show in this chapter that T/TCP behaves differently from TCP. In particular, there are executions of T/TCP where the same data is delivered twice.

T/TCP is an extension of TCP, so the formal model for T/TCP is an extension of the formal model for T/TCP. Thus, is will include many of the same state variables and actions. Recall from our informal description of T/TCP in Chapter 2 that it uses a dual monotonic numbering scheme (connection counts paired with sequence numbers), and persistent caching to accelerate the opening phase of TCP and bypass the three-way handshake protocol. However, if the cache data is lost, T/TCP reverts back to the three-way handshake. Because T/TCP has the TAO mechanism, but must still be able to perform the three-way handshake when necessary, it is more complex that TCP. The TAO mechanism introduces partially synchronized states that do not exist in TCP. Partially synchronized states are states of the server where is has accepted data and is thus synchronized, but the client may not yet be in a state where it can accept data, and is thus unsynchronized. The TAO mechanism also introduces a special state of the client where it send a SYN control

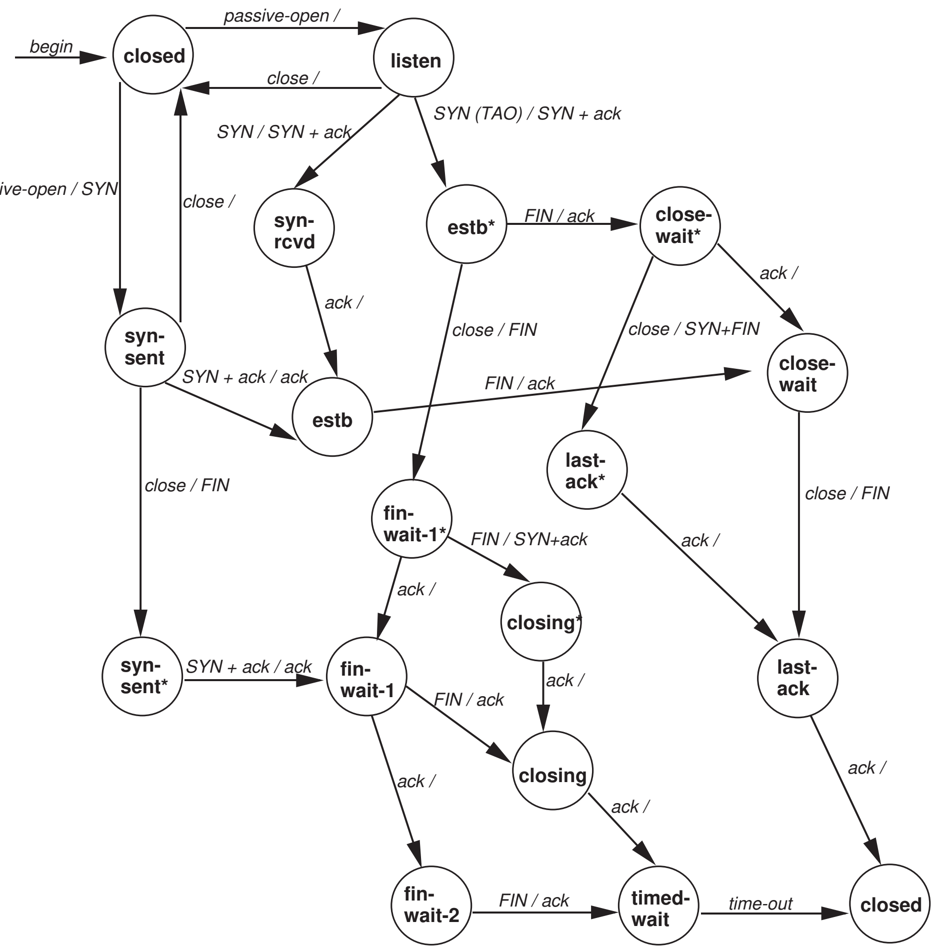

Figure 9-1: *The T/TCP finite state machine. Each host begins in the closed state. Labels on transitions show the input that caused the transition followed by the output if any. The starred states are states not in the TCP FSM.*

signal, send data, and send the FIN control signal all in one segment.

Figure 9-1 shows the T/TCP finite state machine which is an extension of the TCP FSM of Chapter 6. It is a slight simplification of the FSM presented by Braden in [7]. As was the case with the TCP FSM, the T/TCP FSM presented here does not have all the details of the protocol, but is presented as an aid to understanding the steps of the T/TCP general timed automaton. For the presentation given in this chapter we assume that the connection count generator along with the sequence number of the client and the server are unbounded and stable. Other variables that are assigned values based on the connection count generator are also unbounded, but are not stable. Because we assume

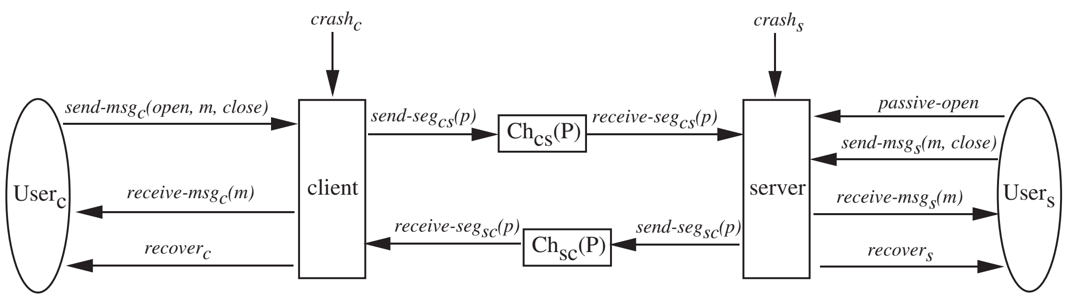

Figure 9-2: *The user interface for T/TCP which is the same as the interface for* TCP

that the connection count generator is stable and unbounded, we do not include in our formal model the special steps taken by the client after a crash to alert the server that the connection count value has been lost due to the crash.

9.1 T/TCP client and server

Like the automaton for TCP, the automaton for T/TCP also has four components — client and server automata and two channels. The structure of the composed system and the user interface is also the same as the structure and user interface for TCP and is shown in Figure 9-2. In this section we present the T/TCP client and server timed automata. We call these timed automata $\mathcal{TTCP}_c$ and $\mathcal{TTCP}_s$ respectively. The channels use for T/TCP are the same as the ones used for TCP. They are presented as timed automata $Ch_{cs}(\mathcal{P})$ and $Ch_{sc}(\mathcal{P})$ in Chapter 5.

9.1.1 States and start states

One of the main difference between TCP and T/TCP is that T/TCP has persistent state. Recall that persistent state means some variables retain their value even when the connection is closed. Persistent state is different from stable state in that persistent variables are affected by crashes, whereas stable variables are not. Thus, in T/TCP when $mode_c$ and $mode_s$ are `closed`, not all the other variables are undefined. In the actual T/TCP protocol, a persistent copy of the last connection count used for a successful connection is kept at the server. However, because we assume the connection count generator is stable and unbounded, we do not have this persistent variable in our model. The only persistent variable we have is $cache\_cc$ on the server side. On the client side cc_gen, now_c, and sn_c are stable, and on the server now_s and sn_s are stable. The other variables on the client are undefined when $mode_c$ is `closed`. Similarly, the non-stable and non-persistent variables on the server side are undefined when $mode_s$ is `closed`. The values given as the initial value for non-stable and non-persistent variables in the tables below are the values they are initialized to when a host opens and initializes its transmission control block. Thus, in the start state of the client side automaton, $\mathcal{TTCP}_c$, $mode_c$ = `closed`, $cc\_gen = 0$, $now_c = 0$, $sn_c = 0$, and all other variables are undefined; and in the start state of the server side automaton, $\mathcal{TTCP}_s$, $mode_s$ = `closed`, $now_c = 0$, $sn_s = 0$, $cache\_cc = \infty$, and all other variables are undefined.

As is the case for the $\mathcal{TCP}$ automaton, we use the set *Msg* to represent the set of possible messages. That is, the set *Msg* is the set of all possible strings over some basic message alphabet that does not include the special symbol `null`. The symbol `null` indicates the absence of a message. The type `T` ranges over the nonnegative real numbers and represents time.

The first table below summarizes the type definitions. In the other tables we describe the variables of $\mathcal{TTCP}_c$ and $\mathcal{TTCP}_s$. A check in the **S** column means the variable is stable and a check in the **P** means the variable is persistent. Because T/TCP is an extension of TCP, the tables below also include many of the same variables defined for $\mathcal{TCP}_c$ and $\mathcal{TCP}_s$. We repeat the descriptions here for completeness.

Type definitions

Type	Description
Msg	The set of all possible strings over some basic message alphabet that does not include the special symbol `null`.
`T`	The nonnegative real numbers — represents real time.
`N`	The set of non-negative integers.

Client variables

Variable	Type	S	P	Initially	Description
$mode_c$	{`closed`, `syn-sent`, `estb`, `fin-wait-1`, `fin-wait-2`, `close-wait`, `last-ack`, `closing`, `timed-wait`, `rec`, `reset`, `syn-sent*`}		$\surd$	`closed`	The modes of the client. The new mode, `syn-sent*` means the client wants to establish a connection, send a message, and send the FIN bit. All the other modes have the same interpretation as in TCP
send-tao-syn	Bool			`false`	A flag that enables the sending of a SYN during TAO even if there is no data to be sent as yet.
tao-syn-sent	Bool			`false`	A flag that indicates the client has started sending SYN segments for TAO.
cc_gen	N	$\surd$		0	The connection count generator.
cc_send	N			0	The connection count for the current incarnation.
sn_c	N	$\surd$		0	Client side sequence number.
msg_c	$Msg \cup \{$`null`$\}$			`null`	The current message to be sent.
$send\text{-}buf_c$	Msg^*			ϵ	The client buffer for messages to be sent.
$rcv\text{-}buf_c$	Msg^*			ϵ	The client buffer for messages received.
ack_c	$N \cup \{$`nil`$\}$			`nil`	The acknowledgment number.
$rst\text{-}seq_c$	$N \cup \{$`nil`$\}$			`nil`	The number assigned to reset segments.
now_c	T	$\surd$		0	The clock variable.
$ready\text{-}to\text{-}send_c$	Bool			`true`	A flag that when true indicates that the next segment can be sent.

Client variables

Variable	Type	S	P	Initially	Description
send-ack$_c$	Bool			`false`	A flag that enables the sending of an acknowledgment.
send-fin$_c$	Bool			`false`	A flag that enables the sending of a *FIN* segment.
rcvd-close$_c$	Bool			`false`	A flag that is set to true when the signal *close* is received.
push-data$_c$	Bool			`false`	A flag that forces the client to only perform the *receive-msg*$_c$*(m)* action until *rcv-buf*$_c$ is empty when a *FIN* segment is received.
send-rst$_c$	Bool			`false`	A flag that enables the sending of a reset segment.
send-fin-ack$_c$	Bool			`false`	A flag that is set to true when the client acknowledges a FIN from mode `closing`.
first(t-out$_c$*)*	$\mathtt{T} \cup \infty$			∞	Used to mark the start of `timed-wait` state.
time-sent$_c$	$\mathtt{T} \cup \infty$			0	Used to mark the time a segment is sent, so that the segment can be resent after RTO if it is not acknowledged.

Server variables

Variable	Type	S	P	Initially	Description
mode$_s$	{`closed`, `listen`, `syn-rcvd`, `estb`, `fin-wait-1`, `fin-wait-2`, `close-wait`, `last-ack`, `timed-wait`, `rec`, `reset`, `estb*`, `fin-wait1*`, `close-wait*`, `closing*`, `last-ack*`}		$\surd$	`closed`	The server modes. The new modes for T/TCP are `estb*`, `fin-wait1*`, `close-wait*`, `closing*`, `last-ack*`. These "starred" modes indicate that the client is in a partially synchronized state. That is, these modes have the same interpretation has their non-starred versions except that the server gets to these modes after a successful TAO which means the client may not have synchronized as yet.
sn$_s$	`N`	$\surd$		0	Server side sequence number.
send-rst$_s$	Bool			`false`	Symmetric to *send-rst*$_c$
now$_s$	`T`	$\surd$		0	The clock variable.
first(t-out$_s$*)*	$\mathtt{T} \cup \infty$			∞	Symmetric to *first(t-out*$_c$*)*.

Server variables

Variable	Type	S	P	Initially	Description
$time\text{-}sent_s$	$\mathtt{T} \cup \infty$			0	Symmetric to$time\text{-}sent_c$.
$send\text{-}buf_s$	Msg^*			ϵ	The buffer for messages to be sent.
$rcv\text{-}buf_s$	Msg^*			ϵ	The buffer for messages received.
ack_s	$\mathtt{N} \cup \{\mathtt{nil}\}$			`nil`	The acknowledgment number.
$rst\text{-}seq_s$	$\mathtt{N} \cup \{\mathtt{nil}\}$			`nil`	Symmetric to $rst\text{-}seq_c$.
$ready\text{-}to\text{-}send_s$	Bool			`true`	Symmetric to $ready\text{-}to\text{-}send_c$.
$send\text{-}ack_s$	Bool			`false`	Symmetric to $send\text{-}ack_c$.
$send\text{-}fin_s$	Bool			`false`	Symmetric to $send\text{-}fin_c$.
$rcvd\text{-}close_s$	Bool			`false`	Symmetric to $rcvd\text{-}close_c$.
$push\text{-}data_s$	Bool			`false`	Symmetric to $push\text{-}data_c$
$send\text{-}fin\text{-}ack_s$	Bool			`false`	Symmetric to $send\text{-}fin\text{-}ack_c$.
cc_rcvd	$\mathtt{N}$			0	The value of the connection count received from the client for the current incarnation.
$cache\_cc$	$\mathtt{N}$		$\surd$	∞	A persistent cached copy of the last connection count number received from the client. The initial value ∞ represents an undefined cache state.
$temp\text{-}data$	$Msg \cup \{\mathtt{null}\}$			`null`	Temporary data variable. This variable stores data that the server cannot accept as yet because the TAO test failed.
$fin\text{-}rcvd$	Bool			`false`	A flag that records that a FIN bit was sent on a segment that failed the TAO test.

9.1.2 Action Signature

Client side, $\mathcal{TTCP}_c$

Input actions:
$send\text{-}msg_c(open,\ m,\ close)$
  $open,\ close \in$ Bool, m $\in$ Msg $\cup\{\mathtt{null}\}$
$receive\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s)$
$receive\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)$
$receive\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s,\ msg_s,\ FIN)$
$receive\text{-}seg_{sc}(cc\_rcvd,\ sn_s,\ ack_s,\ msg_s,)$
$receive\text{-}seg_{sc}(cc\_rcvd,\ sn_s,\ ack_s,\ msg_s,\ FIN)$
$receive\text{-}seg_{sc}(RST,\ ack_s,\ rst\text{-}seq_s)$
$crash_c$
Output actions:
$receive\text{-}msg_c(m),\ m \in Msg$
$send\text{-}seg_{cs}(SYN,\ cc\_send,\ sn_c,\ msg_c)$
$send\text{-}seg_{cs}(SYN,\ cc\_send,\ sn_c,\ msg_c,\ FIN)$
$send\text{-}seg_{cs}(cc\_send,\ sn_c,\ ack_c,\ msg_c)$
$send\text{-}seg_{cs}(cc\_send,\ sn_c,\ ack_c,\ msg_c,\ FIN)$
$send\text{-}seg_{cs}(RST,\ ack_c,\ rst\text{-}seq_c)$
$recover_c$
Internal actions:
$time\text{-}out_c$
$prepare\text{-}msg_c$

$shut\text{-}down_c$
Time-passage actions:
$\nu(t)$, $t \in R^+$

Server side, $\mathcal{TTCP}_s$

Input actions:
passive-open
$send\text{-}msg_s(m,\ close)$
$receive\text{-}seg_{cs}\ (SYN,\ cc\_send,\ sn_c,\ msg_c)$
$receive\text{-}seg_{cs}\ (SYN,\ cc\_send,\ sn_c,\ msg_c,\ FIN)$
$receive\text{-}seg_{cs}\ (cc\_send,\ sn_c,\ ack_c,\ msg_c)$
$receive\text{-}seg_{cs}\ (cc\_send,\ sn_c,\ ack_c,\ msg_c,\ FIN)$
$receive\text{-}seg_{cs}(RST,\ ack_c,\ rst\text{-}seq_c)$
$crash_s$
Output actions:
$receive\text{-}msg_s(m)$, m $\in$ Msg
$send\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s)$
$send\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)$
$send\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s,\ msg_s,\ FIN)$
$send\text{-}seg_{sc}(cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)$
$send\text{-}seg_{sc}(cc\_rcvd,\ sn_s,\ ack_s,\ msg_s,\ FIN)$
$send\text{-}seg_{sc}(RST,\ ack_s,\ rst\text{-}seq_s)$
$recover_s$
Internal actions:
$time\text{-}out_s$
$prepare\text{-}msg_s$
$shut\text{-}down_s$
Time-passage:
$\nu(t)$, $t \in R^+$

9.1.3 Steps of T/TCP

The steps for the timed automata for T/TCP are shown in Figures 9-3, 9-4, 9-5, 9-6, 9-7, 9-8, 9-9, and 9-10. In all the figures the steps of the client automaton, $\mathcal{TTCP}_c$ is on the left and the steps of the server automaton, $\mathcal{TTCP}_s$ is on the right. The *receive-seg(p)* actions are shown opposite the corresponding send *send-seg(p)* actions, and symmetric internal actions are also opposite each other. Because there are many possible ways an execution can proceed, the steps of the protocol presented in the figures should not be thought of as being in a sequential order. Instead, the appropriate steps are referred to as possible executions are described.

In T/TCP if the TAO mechanism works, there may not be distinct open, data transfer, and close phases for a connection. If the TAO mechanism fails, then T/TCP has these three phases, and the open phase is the three-way handshake protocol. The description of the steps of the automata presented here is divided into two cases based on whether the TAO mechanism fails or not. However, both cases have commonalities which we present first.

A connection is always started in the same manner. That is, the client gets the *send-msg$_c$(open, m, close)* input when $mode_c$ is `closed`, and the server gets *passive-open* when $mode_s$ is `closed` (Figure 9-3). If *open* is true on the client side, then the transmission control block is initialized, *cc_gen* and sn_c are incremented, *cc_send* is assigned the value of *cc_gen*, and $mode_c$ is set to `syn-sent`. On the server side TCB_s is initialized and $mode_s$ gets set to `listen`.

As is the case with $\mathcal{TCP}$ the retransmission of segments is determined by the retransmission timeout (RTO). Therefore, all the $send\text{-}seg_{cs}(p)$ actions have as a part of the precondition that $now_c - time\text{-}sent_c$ is greater that or equal to RTO. This precondition and the setting of $time\text{-}sent_c$ to the value of now_c in the effect clause of these actions means the actions are not enabled again until RTO passes or an acknowledgment is received. Because, of this precondition in the $send\text{-}seg_{cs}(p)$ actions, in the $receive\text{-}seg_{sc}(p)$ actions if the incoming segment is a segment that needs to be acknowledged, that is, if the sequence number, $[sn_s]$, on the incoming segment is greater than or equal to the acknowledgment number ack_c of the client, $time\text{-}sent_c$ is set to 0. The setting of $time\text{-}sent_c$ to 0 allows the first transmission of a response segment without the RTO delay. On the server side, the $send\text{-}seg_{sc}(p)$ and $receive\text{-}seg_{cs}(p)$ actions are symmetric.

The basic mechanism for acknowledging segments is also the same as it is in TCP. That is, to acknowledge a segment with sequence number i, a host returns a segment with the ack_c or ack_s variable set to $i+1$. In all the $receive\text{-}seg_{sc}(p)$ and $receive\text{-}seg_{cs}(p)$ actions, if segment p is an old duplicate and the receiving host is in an unsynchronized state, or if the receiving host is closed, then assignments are made to generate a reset segment. Namely, the $send\text{-}rst_c$ or $send\text{-}rst_s$ is set to true, and the $rst\text{-}seq_c$ or $rst\text{-}seq_s$ is set to either 0 or $[ack_s]$ or $[ack_c]$ respectively.

The steps with other actions in the $\mathcal{TTCP}_c$ and $\mathcal{TTCP}_s$ general timed automata, such as $time\text{-}out_c$, $time\text{-}out_s$, $crash_c$ and $crash_s$; and $recover_c$ (Figure 9-9) are the same as for the TCP timed automaton described in Section 6.1.3. The $recover_s$ step is slightly different in that *cache_cc* is assigned ∞ in this step. The steps with the reset and the shut down actions (Figure 9-10); and the time passage actions (Figure 9-5) are also the same as for the TCP timed automaton. The reader is referred to Chapter 6 for a description of these steps.

TAO test fails

When the TAO test fails, the protocol reverts to the three-way handshake to synchronize the end-points. That is, the protocol behaves like TCP in the open phase. We describe the workings of the formal model in this situation. When the client receives the $send\text{-}msg_c(open, m, close)$ input action (Figure 9-3), if $mode_c =$ `closed` and *open* is *true*, it prepares to send a SYN segment. If $m \neq$ `null`, it is added to the send buffer and $ready\text{-}to\text{-}send_c$ is assigned to `false` in order to enable the $prepare\text{-}msg_c$ action and *send-tao-syn* is set to `false` to ensure that the SYN segment is not sent until the message is ready. This is the only situation where the $prepare\text{-}msg_c$ action is enabled as a consequence of the $send\text{-}msg_c(open, m, close)$ action. If the client has already sent a SYN segment when the m is received, that is, *tao-syn-sent* = `true`, then the prepare message action is not enabled until a response is received. Likewise, if there is no message to send, $m =$ `null`, then the $prepare\text{-}msg_c$ action should not be enabled. It is for these reasons that the assignment of $ready\text{-}to\text{-}send_c$ is conditioned on m not being `null` and *tao-syn-sent* being false.

If the $prepare\text{-}msg_c$ action is enabled (Figure 9-4) it means there is data to send in SYN segment. In this step sn_c gets incremented, $ready\text{-}to\text{-}send_c$ is set to `true`, and msg_c is assigned to the head of $send\text{-}buf_c$. If the close signal had been received and $send\text{-}buf_c = \epsilon$, the sequence number is incremented again to note the sending of a FIN segment, $ready\text{-}to\text{-}send_c$ is set to `false` to disable the sending of the segment without the FIN signal while $send\text{-}fin_c$ is set to true to enable the sending of the FIN segment. Since we are describing the opening phase of an execution, in this situation $mode_c$ is `syn-sent`, so it would get set to `syn-sent*`.

If the closed signal was not received, then if there was no data to send, or if there was data then after the $prepare\text{-}msg_c$ action, the $send\text{-}seg_{cs}(SYN, sn_c, ack_c, msg_c)$ action (Figure 9-3) is performed. If there was data to be sent, and the close signal was also received, the $send\text{-}seg_{cs}(SYN, sn_c, ack_c, msg_c, FIN)$ action (Figure 9-8) is performed. When the server receives the $(SYN, sn_c, ack_c, msg_c)$ segment (Figure 9-3) if it is in mode `listen`, it firsts assigns *cc_rcvd* to *cc_send*, increments its sequence number, and make the assignments necessary for an acknowledgment segment. Since we are examining an execution where the TAO test fails, either *cache_cc* is undefined or *cache_cc* $\geq$ *cc_send*. When this happens $mode_s$ is set to `syn-rcvd`, *cache_cc* is reset to 0, and *temp-data* is assigned msg_c. If the *(SYN,* sn_c*,* ack_c*,* msg_c*, FIN)* segment is received by the server instead, the same assignments are made and additionally *fin-rcvd* is set to `true` to indicate that a FIN has been received.

With $mode_s$ = `syn-rcvd` the next action enabled on the server side is *send-seg*$_{sc}$*(SYN, cc_rcvd,* sn_s*,* ack_s*)* (Figure 9-4). When the client receives this segment, if a reset is not generated, it sets *send-ack*$_c$ to `true` to enable the acknowledgment of this segment. If the client is already in a synchronized state, this segment is either a duplicate created by the channel or a retransmission of a segment previously acknowledged. Since the acknowledgment might not have been received by the server, the retransmission of the acknowledgment is enabled. If $mode_c \in \{$`syn-sent`, `syn-sent*`$\}$, and $ack_s = sn_c + 1$, then the client knows the server received its correct initial sequence number, and that the segment is an acknowledgment of the SYN segment it sent. It also sets msg_c to `null` to indicate the acknowledgment of that message. The client also changes mode to either `estb` or `fin-wait-1`, and prepares to send a response. First ack_c is set to $sn_s + 1$ for the next expected segment, and *time-sent*$_c$ gets set to 0. Then if there is data to be sent, the flag *ready-to-send*$_c$ is set `false` to enable the *prepare-msg*$_c$ action.

The final part of the three-way handshake is the action *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* (Figure 9-6) or if the client had received a *close* input and had no more data to send, *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)* (Figure 9-7). Both segments acknowledge the SYN segment from the server. In the open phase, when the server receives either of these segments it is in state `syn-rcvd`. It first checks that they are not old duplicates, that is, if [*cc_send*] is equal to *cc_rcvd*, and $[ack_c] = sn_s + 1$. If the segment in either action is valid the server can now update *cache_cc*, so it sets it to *cc_send*. If *temp-data* is not `null` it is concatenated to *rcv-buf*$_s$, and then set to `null`. For the *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* step, $mode_s$ is set to `close-wait` if *fin-rcvd* is `true`, and `estb` if it is `false`. For the *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)* step, $mode_s$ is set to `close-wait`. The remaining assignments adds [msg_c] to the receive buffer if it is not `null` and enables the *prepare-msg*$_s$ (Figure 9-4) action if there the server has data. The client and server are now both in synchronized states and data transfer either starts or continues.

The data transfer phase of T/TCP is very much like the data transfer phase of TCP described in Chapter 6. The client gets data from the user via the *send-msg*$_c$*(open, m, close)* input action and the server gets data via the *send-msg*$_c$*(m, close)* action. The client prepares to send data in the *prepare-msg*$_c$ action, and sends data and acknowledgments with the *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* or the *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)* actions. The server side is almost symmetric: it prepares to send data in the *prepare-msg*$_c$ action and sends data and acknowledgments with the *send-seg*$_{sc}$*(cc_rcvd, sn*$_s$*, ack*$_s$*, msg*$_s$*)* action (Figure 9-9) and the *send-seg*$_{sc}$*(cc_rcvd, sn*$_s$*, ack*$_s$*, msg*$_s$*, FIN)* action (Figure 9-10) when it is ready to close. Data received by the client is passed to the user by the *receive-msg*$_c$$(m)$ action (Figure 9-5). The symmetric action of the server side is *receive-msg*$_s$$(m)$ (Figure 9-5).

The close phase of T/TCP is also like the close phase of TCP. The main difference is that the server has several partially synchronized modes now associated with the close phase (`fin-wait1*`, `close-wait*`, `closing*`, `last-ack*`). However, in executions where the TAO mechanism does not work, the server does not get to these partially synchronized modes. Either side can begin the close phase. A host starts the close phase when it receives a close signal from the user, *send-msg*$_c$*(open, m, close)* on the client side and *send-msg*$_s$*(m, close)* on the server side with *close* true for both, or when it receives a FIN segment from the other host. When the client receives the signal to close, it sets *rcvd-close*$_s$ to `true`, but it does not start the close phase until it sends all the messages that are in its send buffer. The client prepares to send a FIN segment by incrementing *sn*$_c$, in the *prepare-msg*$_c$ action, once if the FIN segment does not have valid data or twice if it does. In that action *send-fin*$_c$ is also set to `true` to enable the sending of the FIN segment, and the mode of the client is set to `fin-wait-1` if it was previously `estb`, or if the client had already received a FIN segment from the server and thus was in mode `close-wait`, it changes to mode `last-ack`. The action at the client to send the FIN segment is *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)*. When the server receives this segment its actions are almost the same as when it receives the data segment *(sn*$_c$*, ack*$_c$*, msg*$_c$*)* except now the message is valid only if $[sn_c] = ack_s + 1$, and it also changes mode. The server responds with the *send-seg*$_{sc}$*(cc_rcvd, sn*$_s$*, ack*$_s$*, msg*$_s$*)* action if it is just acknowledging the FIN segment, or it can acknowledge the FIN segment and send its own FIN with the *send-seg*$_{sc}$*(cc_rcvd, sn*$_s$*, ack*$_s$*, msg*$_s$*, FIN)*. The hosts close when one or both sides *timeout* after waiting $2 \times \mu$ in mode `timed-wait`, and if only one side closes from `timed-wait` state, the other closes after receiving an acknowledgment while in mode `last-ack`.

TAO test succeeds

In executions where the TAO mechanism works, there are three possibilities for the SYN segment the client sends. The segment can either have no valid data, or have valid data, but no FIN bit, or it can have valid data and a FIN bit. Similarly, the server can respond with a SYN segment that contains no valid data, valid data without a FIN, or valid data and a FIN. In a typical transaction, that is, one where the client sends one piece of request data and gets one piece of response data in return, the first segment the client sends has valid data and a FIN bit, and the server responds with a similar segment. We describe the sequence of steps in $\mathcal{TTCP}_c$ and $\mathcal{TTCP}_s$ for such an execution. On the client side the execution starts exactly as executions where TAO does not work, described above. The differences start when the server compares *cache_cc* to [*cc_send*] in the *receive-seg*$_{cs}$*(SYN, sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)* step (Figure 9-8). Here TAO is successful because *cache_cc* $<$ [*cc_send*], so *cache_cc* is updated to *cc_send*. Then since $msg_c \neq$ `null`, the message is added to the receive buffer of the server. Next *mode*$_s$ is assigned the value `close-wait*`, and *push-data*$_s$ is set to true to ensure that the data is passed to the user before the server closes. The data is passed to the server side user with the *receive-msg*$_s$*(m)* action (Figure 9-5), and since it is the only piece of data, the receive buffer becomes empty after it is passed to the user, so *push-data*$_s$ is set to `false`. For the type of execution we are describing, after it passes the data to the user, the server gets response data and the signal to close in the *send-msg*$_s$*(m, close)* input action. Now the *prepare-msg*$_s$ action (Figure 9-4) is enabled. In this action the message is removed from the send buffer to the *msg*$_s$ variable, *sn*$_s$ is incremented twice, *send-fin*$_s$ is set to `true`, and mode$_s$ changes from `close-wait*` to `last-ack*`. Next the server performs the the *send-seg*$_{sc}$*(SYN, cc_rcvd, sn*$_s$*, ack*$_s$*, msg*$_s$*, FIN)* action (Figure 9-10). When the client receives the segment, it checks that *cc_rcvd* is equal to *cc_send*. If it is, the client changes form mode `syn-sent*` to `timed-wait`, assigns *push-data*$_c$ to `true`, puts the message on its receive buffer, and make other assignments to send an acknowledgment. The client passes the data to its user with the *receive-msg*$_c$*(m)* action (Figure 9-5) which also sets *push-data*$_c$ back to `false`. With *push-data*$_c$ `false`, the *send-seg*$_{cs}$*(cc_send, sn*$_c$*, ack*$_c$*, msg*$_c$*)* action (Figure 9-6) is now enabled. When the client performs this action it starts the timer for `timed-wait` state. When the server receives the segment with the *receive-seg*$_{cs}$*(cc_send, sn*$_c$*, ack*$_c$*, msg*$_c$*)* action, it changes from mode `last-ack*` to `closed`. After waiting for $2 \times \mu$, the client closes with the internal action *time-out*$_c$.

Several other variations of execution sequences are possible when the TAO mechanism works. For example, the TAO mechanism may work, and the client user wants to send more than one piece of data, so it does not send the signal to close immediately . In this case the connection has a data transfer phase and a close phase that is essentially the same as the data transfer phase and close phase when TAO does not work. Another possible variation is that the client user may just want to send one piece of request data, but the server user has lots of response data. For such a scenario, again there is a data transfer and close phase. All the other possibilities are variations on the execution sequences we describe above, so we do not describe them here.

send-msg$_c$*(open, m, close)*
Eff: if $mode_c$ = closed $\wedge$ *open* then {
initialize TCB_c
$time\text{-}sent_c := 0$
$cc\_gen := cc\_gen + 1$
$sn_c := sn_c + 1$
$cc\_send := cc\_gen$
$mode_c :=$ syn-sent
$send\text{-}tao\text{-}syn :=$ true
}
if $mode_c \in$ {syn-sent, estb, close-wait}
$\wedge \neg$ $rcvd\text{-}close_c \wedge$ $m \neq$ null then {
$send\text{-}buf_c := send\text{-}buf_c \cdot m$
if $mode_c$ = syn-sent $\wedge$ m = null $\wedge$
$\neg tao\text{-}syn\text{-}sent$ then {
$ready\text{-}to\text{-}send_c :=$ false
$send\text{-}tao\text{-}syn :=$ false
}
}
if *close* then {
$rcvd\text{-}close_c :=$ true
if $mode_c$ = syn-sent $\wedge$ msg_c = null $\wedge$
$\neg tao\text{-}syn\text{-}sent$ $\wedge$ $send\text{-}buf_c = \epsilon$ then
$mode_c :=$ closed
}

send-seg$_{cs}$*(SYN, cc_send, sn*$_c$*, msg*$_c$*)*
Pre: $(now_c - time\text{-}sent_c \geq \mathrm{RTO}) \wedge$
$\wedge(ready\text{-}to\text{-}send_c \vee send\text{-}tao\text{-}syn)$
$mode_c$ = syn-sent $\wedge \neg send\text{-}rst_c$
Eff: $time\text{-}sent_c := now_c$
$tao\text{-}syn\text{-}sent :=$ true

passive-open
Eff: if $mode_s$ = closed then {
initialize TCB_s
$mode_s :=$ listen
}

send-msg$_s$*(m, close)*
Eff: if $mode_s \in$ {estb, estb*,
close-wait, close-wait*} $\wedge$
$\neg$ $rcvd\text{-}close_s \wedge m \neq$ null then
$send\text{-}buf_s := send\text{-}buf_s \cdot m$
if *close* then {
$rcvd\text{-}close_s :=$ true
else if $mode_s$ = listen $\wedge$
$send\text{-}buf_s = \epsilon$ then
$mode_s :=$ closed
}

receive-seg$_{cs}$ *(SYN, cc_send, sn*$_c$*, msg*$_c$*)*
Eff: if $mode_s$ = listen then {
$cc\_rcvd := cc\_send$
$sn_s := sn_s + 1$
$ack_s := sn_c + 1$
$time\text{-}sent_s := 0$
$send\text{-}ack_s :=$ true
if $cache\_cc < cc\_send$ then {
$cache\_cc := cc\_send$
$ready\text{-}to\text{-}send_s :=$ false
if $msg_c \neq$ null then
$rcv\text{-}buf_s := rcv\text{-}buf_s \cdot m$
$mode_s :=$ estb*
if $send\text{-}buf_s \neq \epsilon$ then
$send\text{-}ack_s :=$ false
}
else {
$mode_s :=$ syn-rcvd
$cache\_cc := \infty$
$temp\text{-}data := msg_c$
}
}
if $mode_s$ = closed then
$send\text{-}rst_s :=$ true
$rst\text{-}seq_s := 0$

Figure 9-3: *Steps from the open phase of* $\mathcal{TTCP}_c$ *and* $\mathcal{TTCP}_s$*. The client steps are on the left and the corresponding server steps are on the right.*

receive-seg$_{sc}$(SYN, cc_rcvd, sn_s, ack_s)
Eff: if ($mode_c$ = closed) $\vee$ ($mode_c \in$ {syn-sent, syn-sent*} $\wedge$ $cc\_rcvd \neq cc\_send$) then {
  $send\text{-}rst_c$:= true
  $rst\text{-}seq_c$:= ack_s
}
else {
  $send\text{-}ack_c$:= true
  if $mode_c \in$ {syn-sent, syn-sent*} then {
    $ack_c := sn_s + 1$
    $time\text{-}sent_c := 0$
    msg_c := null
    $ready\text{-}to\text{-}send_c$:= false
    if $mode_c$ = syn-sent then
      $mode_c$:= estb
    if $mode_c$ = syn-sent* then
      $mode_c$:= fin-wait-1
    if $send\text{-}buf_c \neq \epsilon$ then
      $send\text{-}ack_c$:= false
  }
}

prepare-msg$_c$
Pre: $\neg push\text{-}data_c \wedge \neg ready\text{-}to\text{-}send_c \wedge$
$mode_c \in$ {syn-sent, estb, close-wait}
$\wedge$ ($send\text{-}buf_c \neq \epsilon \vee rcvd\text{-}close_c$)
Eff: $ready\text{-}to\text{-}send_c$:= true
if $send\text{-}buf_c \neq \epsilon$ then {
  $sn_c := sn_c + 1$
  msg_c := head($send\text{-}buf_c$)
  $send\text{-}buf_c$:= tail($send\text{-}buf_c$)
}
if $rcvd\text{-}close_c \wedge send\text{-}buf_c = \epsilon$ then {
  $sn_c := sn_c + 1$
  $ready\text{-}to\text{-}send_c$:= false
  $send\text{-}fin_c$:= true
  if mode$_c$ = syn-sent then
    mode$_c$:= syn-sent*
  if $mode_c$ = estb then
    $mode_c$:= fin-wait-1
  if $mode_c$ = close-wait then
    $mode_c$:= last-ack
}

send-seg$_{sc}$(SYN, cc_rcvd, sn_s, ack_s)
Pre: (now_s - $time\text{-}sent_s \geq$ RTO) $\wedge$
$mode_s$ = syn-rcvd $\wedge \neg$ $send\text{-}rst_s$
Eff: $time\text{-}sent_s := now_s$

prepare-msg$_s$
Pre: $\neg push\text{-}data_s \wedge \neg ready\text{-}to\text{-}send_s \wedge$
($send\text{-}buf_s \neq \epsilon \vee rcvd\text{-}close_s$) $\wedge$
$mode_s \in$ {estb, estb*, close-wait, close-wait*}
Eff: $ready\text{-}to\text{-}send_s$:= true
if $send\text{-}buf_s \neq \epsilon$ then {
  $sn_s := sn_s + 1$
  msg_s := head($send\text{-}buf_s$)
  $send\text{-}buf_s$:= tail($send\text{-}buf_s$)
}
if $rcvd\text{-}close_s \wedge send\text{-}buf_s = \epsilon$ then {
  $sn_s := sn_s + 1$
  $ready\text{-}to\text{-}send_s$:= false
  $send\text{-}fin_s$:= true
  if $mode_s$ = estb then
    $mode_s$:= fin-wait-1
  if $mode_s$ = estb* then
    $mode_s$:= fin-wait1*
  else $mode_s$ = close-wait then
    $mode_s$:= last-ack
  else $mode_s$ = close-wait* then
    $mode_s$:= last-ack*
}

Figure 9-4: *The first pair of steps complete the three-way handshake in T/TCP and the next pair are the steps that prepare messages to be sent.*

receive-seg$_{sc}$(SYN, cc_rcvd, sn_s, ack_s, msg_s) `Eff:` if $mode_c =$ `closed` $\vee$ $(cc\_rcvd_s \neq cc\_send_c$ $\wedge\ mode_c \in \{$`syn-sent`, `syn-sent*`$\})$ then { $send\text{-}rst_c :=$ `true` $rst\text{-}seq_c := ack_s$ } else { $send\text{-}ack_c :=$ `true` $mode_c \in \{$`syn-sent`, `syn-sent*`$\}$ then { $ack_c := sn_s + 1$ $time\text{-}sent_c := 0$ $msg_c :=$ `null` $ready\text{-}to\text{-}send_c :=$ `false` $send\text{-}fin_c :=$ `false` if $mode_c =$ `syn-sent` then $mode_c :=$ `estb` if $mode_c =$ `syn-sent*` then $mode_c :=$ `fin-wait-2` if $msg_s \neq$ `null` then $rcv\text{-}buf_c := rcv\text{-}buf_c \cdot msg_s$ if $send\text{-}buf_c \neq \epsilon$ then $send\text{-}ack_c :=$ `false` } }	*send-seg$_{sc}$(SYN, cc_rcvd, sn_s, ack_s, msg_s)* `Pre:` $(now_s - time\text{-}sent_s \geq \mathrm{RTO}) \wedge$ $mode_s \in \{$`estb*`, `close-wait*`$\} \wedge$ $(ready\text{-}to\text{-}send_s \vee send\text{-}ack_s) \wedge$ $\neg push\text{-}data_s$ `Eff:` $time\text{-}sent_s := now_s$
receive-msg$_c$(m) `Pre:` $mode_c \notin \{$`rec`, `reset`$\} \wedge rcv\text{-}buf_c \neq \epsilon$ $\wedge\ \mathrm{head}(rcv\text{-}buf_c) = m$ `Eff:` $rcv\text{-}buf_c := \mathrm{tail}(rcv\text{-}buf_c)$ if $push\text{-}data_c \wedge rcv\text{-}buf_c = \epsilon$ then $push\text{-}data_c :=$ `false`	*receive-msg$_s$(m)* `Pre:` $mode_s \notin \{$`rec`, `reset`$\} \wedge rcv\text{-}buf_s \neq \epsilon$ $\wedge head(rcv\text{-}buf_s) = m$ `Eff:` $rcv\text{-}buf_s := \mathrm{tail}(rcv\text{-}buf_s)$ if $push\text{-}data_s \wedge rcv\text{-}buf_s = \epsilon$ then $push\text{-}data_s :=$ `false`
$\nu(t)$ *(time-passage)* `Pre:` $t \in R^+$ `Eff:` $now_c := now_c + t$	$\nu(t)$ (time-passage) `Pre:` $t \in R^+$ `Eff:` $now_s := now_s + t$

Figure 9-5: *The steps in which the server sends a SYN segment in response to a successful TAO, and the receiving of that segment on the client side. The second pair of steps pass messages to the users, and the third pair are the time-passage actions.*

send-seg$_{cs}$*(cc_send, sn*$_c$*, ack*$_c$*, msg*$_c$*)*

Pre: $(now_c - time\text{-}sent_c \geq RTO) \wedge$
$((ready\text{-}to\text{-}send_c \vee send\text{-}ack_c) \wedge$
$mode_c \in sync\text{-}states \wedge \neg push\text{-}data_c$

Eff: $time\text{-}sent_c := now_c$
$send\text{-}ack_c :=$ `false`
if $mode_c =$ `closing` then
  $send\text{-}fin\text{-}ack_c :=$ `true`
if $mode_c =$ `timed-wait` then {
  $first(t\text{-}out_c) := now_c$
  $time\text{-}sent_c := \infty$
}

receive-seg$_{cs}$*(cc_send, sn*$_c$*, ack*$_c$*, msg*$_c$*)*

Eff: if $(mode_s =$ `syn-rcvd` $\wedge$
$(cc\_send \neq cc\_rcvd \;\vee\; ack_c \neq sn_s + 1)) \vee$
$mode_s \in$ {`closed`, `listen`} then {
  $send\text{-}rst_s :=$ `true`
  $rst\text{-}seq_s := ack_c$
}
else if $mode_s \notin$ {`rec`, `reset`} $\wedge$
$cc\_send = cc\_rcvd$ then {
  if $msg_c \neq$ `null` then
    $send\text{-}ack_s :=$ `true`
  if $ack_c = sn_s + 1$ then {
    $msg_s :=$ `null`
    $ready\text{-}to\text{-}send_s :=$ `false`
    $send\text{-}fin_s :=$ `false`
    if $mode_s =$ `syn-rcvd` then {
      $cache\_cc := cc\_send$
      if $temp\text{-}data \neq$ `null` then {
        $rcv\text{-}buf_s := rcv\text{-}buf_s \cdot temp\text{-}data$
        $temp\text{-}data =$ `null`
      }
      if $fin\text{-}rcvd$ then
        $mode_s :=$ `close-wait`
      else $mode_s :=$ `estb`
    }
    if $send\text{-}buf_s \neq \epsilon$ then
      $send\text{-}ack_s :=$ `false`
    if $mode_s =$ `estb*`
      then $mode_s :=$ `estb`
    if $mode_s \in$ {`fin-wait-1`, `fin-wait1*`}
      then $mode_s :=$ `fin-wait-2`
    if $mode_s \in$ {`last-ack`, `last-ack*`}
      then $mode_s :=$ `closed`
    if $mode_s \in$ {`closing`, `closing*`} then {
      $mode_s :=$ `timed-wait`
      if $send\text{-}fin\text{-}ack_s$ then
        $first(t\text{-}out_s) := now_s$
    }
  }
  if $sn_c = ack_s$ then {
    $ack_s := sn_c + 1$
    $time\text{-}sent_s := 0$
    $rcv\text{-}buf_s := rcv\text{-}buf_s \cdot msg_c$
  }
}

Figure 9-6: *The basic message sending step for* $\mathcal{TTCP}_c$ *is on the left and the corresponding step to receive the segment for* $\mathcal{TTCP}_s$ *is on the right.*

send-seg$_{cs}$*(cc_send, sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)*	*receive-seg*$_{cs}$*(cc_send, sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)*
`Pre:` $(now_c - time\text{-}sent_c \geq \mathrm{RTO}) \wedge$ $send\text{-}fin_c \wedge \neg pdata_c \ \wedge mode_c \in$ $\{$`fin-wait-1`, `last-ack`, `closing`$\}$ `Eff:` $time\text{-}sent_c := now_c$	`Eff:` if $(mode_s =$ `syn-rcvd` $\wedge$ $(cc\_send \neq cc\_rcvd \ \vee \ ack_c \neq sn_s + 1)) \vee$ $mode_s \in$ $\{$`closed`, `listen`$\}$ then { $send\text{-}rst_s :=$ `true` $rst\text{-}seq_s := ack_c$ } else if $mode_s \notin$ $\{$`rec`, `reset`$\}$ $\wedge$ $cc\_send = cc\_rcvd$ then { if $msg_c \neq$ `null` then $send\text{-}ack_s :=$ `true` if $sn_c = ack_s \vee sn_c = ack_s + 1$ then { $push\text{-}data_s :=$ `true` $time\text{-}sent_s := 0$ if $ack_c = sn_s + 1$ then { $msg_s :=$ `null` $ready\text{-}to\text{-}send_s :=$ `false` $send\text{-}fin_s :=$ `false` if $mode_s =$ `syn-rcvd` then { $cache\_cc := cc\_send$ $mode_s :=$ `close-wait` if $temp\text{-}data \neq$ `null` then { $rcv\text{-}buf_s := rcv\text{-}buf_s \cdot temp\text{-}data$ $temp\text{-}data =$ `null` } } if $send\text{-}buf_s \neq \epsilon$ then $send\text{-}ack_s :=$ `false` if $mode_s =$ `estb*` then $mode_s :=$ `estb` if $mode_s \in$ $\{$`fin-wait-1`, `fin-wait1*`$\}$ then $mode_s :=$ `fin-wait-2` } if $sn_c = ack_s + 1$ then $rcv\text{-}buf_s := rcv\text{-}buf_s \cdot msg_s$ $ack_s := sn_c + 1$ if $mode_s =$ `estb` then $mode_s :=$ `close-wait` if $mode_s =$ `fin-wait-1` then $mode_s :=$ `closing` if $mode_s =$ `fin-wait-2` then $mode_s :=$ `timed-wait` } }

Figure 9-7: *The basic step for sending a FIN segment for $\mathcal{TTCP}_c$ is on the left and the corresponding step to receive the segment for $\mathcal{TTCP}_s$ is on the right.*

send-seg$_{cs}$(SYN, cc_send, sn$_c$, msg$_c$, FIN)
Pre: $(now_c - time\text{-}sent_c \geq \text{RTO}) \wedge send\text{-}fin_c$
$\wedge\ mode_c = \texttt{syn-sent*} \wedge cc\ \wedge \neg send\text{-}rst_c$
Eff: $time\text{-}sent_c := now_c$

receive-seg$_{sc}$–
(SYN, cc_rcvd, sn$_s$, ack$_s$, msg$_s$, FIN)
Eff: if $cc\_rcvd\ = cc\_send\ \wedge$
$mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\}$ then {
$send\text{-}ack_c := \texttt{true}$
$ack_c := sn_s + 1$
$time\text{-}sent_c := 0$
$msg_c := \texttt{null}$
$ready\text{-}to\text{-}send_c := \texttt{false}$
$push\text{-}data_c := \texttt{true}$
$send\text{-}fin_c := \texttt{false}$
if $mode_c = \texttt{syn-sent}$ then
$mode_c := \texttt{close-wait}$
if $mode_c = \texttt{syn-sent*}$ then
$mode_c := \texttt{timed-wait}$
if $msg_s \neq \texttt{null}$ then
$rcv\text{-}buf_c := rcv\text{-}buf_c \cdot msg_s$
if $send\text{-}buf_c \neq\ \epsilon$ then
$send\text{-}ack_c := \texttt{false}$
}
if $mode_c = \texttt{closed}\ \vee (cc\_rcvd_s \neq cc\_send_c$
$\wedge\ mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\})$ then
$send\text{-}rst_c := \texttt{true}$
$rst\text{-}seq_c := ack_s$

receive-seg$_{cs}$(SYN, cc_send, sn$_c$, msg$_c$, FIN)
Eff: if $mode_s = \texttt{listen}$ then {
$cc\_rcvd := cc\_send$
$sn_s := sn_s + 1$
$ack_s := sn_c + 1$
$time\text{-}sent_s := 0$
$send\text{-}ack_s := \texttt{true}$
if $cache\_cc\ < cc\_send$ then {
$cache\_cc := cc\_send$
$ready\text{-}to\text{-}send_s := \texttt{false}$
if $msg_c \neq \texttt{null}$ then
$rcv\text{-}buf_s := rcv\text{-}buf_s \cdot msg_c$
$mode_s := \texttt{close-wait*}$
$push\text{-}data_s := \texttt{true}$
if $send\text{-}buf_s \neq\ \epsilon$ then
$send\text{-}ack_s := \texttt{false}$
else {
$mode_s := \texttt{syn-rcvd}$
$cache\_cc := \infty$
$temp\text{-}data := msg_c$
$fin\text{-}rcvd := \texttt{true}$
}
}
if $mode_s = \texttt{closed}$ then
$send\text{-}rst_s := \texttt{true}$
$rst\text{-}seq_s := 0$

send-seg$_{sc}$–
(SYN, cc_rcvd, sn$_s$, ack$_s$, msg$_s$, FIN)
Pre: $(now_s$ - $time\text{-}sent_s \geq \text{RTO}) \wedge$
$send\text{-}fin_s \wedge \neg push\text{-}data_s$
$\wedge mode_s \in \{\texttt{fin-wait1*}, \texttt{last-ack*}\}$
Eff: $time\text{-}sent_s := now_s$

Figure 9-8: *Steps for T/TCP accelerated open. The client (on the left) sends the SYN, data, and FIN in one segment. The server responds with a segment that contains an echo of the cc value the client sends.*

$receive\text{-}seg_{sc}(cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)$
Eff: if $mode_c \in \{$closed, syn-sent, syn-sent*$\}$
  then {
    $send\text{-}rst_c :=$ true
    $rst\text{-}seq_c := ack_s$
  }
  else if $mode_c \notin \{$rec, reset$\} \wedge$
  $cc\_rcvd_s = cc\_send_c$ then {
    if $msg_s \neq$ null then
      $send\text{-}ack_c :=$ true
    if $sn_s = ack_c$ then {
      $ack_c := sn_s + 1$
      $time\text{-}sent_c := 0$
      $rcv\text{-}buf_c := rcv\text{-}buf_c \cdot msg_s$
    }
    if $ack_s = sn_c + 1$ then {
      $msg_c :=$ null
      $ready\text{-}to\text{-}send_c :=$ false
      $send\text{-}fin_c :=$ false
      if $send\text{-}buf_c \neq \epsilon$ then
        $send\text{-}ack_c :=$ false
      if $mode_c =$ fin-wait-1 then
        $mode_c :=$ fin-wait-2
      if $mode_c =$ last-ack then
        $mode_c :=$ closed
      if $mode_c =$ closing then {
        $mode_c :=$ timed-wait
        if $send\text{-}fin\text{-}ack_c$ then
          $first(t\text{-}out_c) := now_c$
      }
    }
  }

$time\text{-}out_c$
Pre: $mode_c =$ timed-wait $\wedge$
  $now_c - first(t\text{-}out_c) \geq 2 \times \mu$
Eff: $mode_c :=$ closed

$crash_c$
Eff: if $mode_c \neq$ closed then
    $mode_c :=$ rec

$recover_c$
Pre: $mode_c =$ rec
Eff: $mode_c :=$ closed

$send\text{-}seg_{sc}(cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)$
Pre: $(now_s - time\text{-}sent_s \geq RTO) \wedge$
  $((ready\text{-}to\text{-}send_s \vee send\text{-}ack_s) \wedge$
  $mode_s \in sync\text{-}states \wedge \neg push\text{-}data_s$
Eff: $time\text{-}sent_s := now_s$
  $send\text{-}ack_s :=$ false
  if $mode_s =$ closing then
    $send\text{-}fin\text{-}ack_s :=$ true
  if $mode_s =$ timed-wait then {
    $first(t\text{-}out_s) := now_s$
    $time\text{-}sent_s := \infty$
  }

$time\text{-}out_s$
Pre: $mode_s =$ timed-wait $\wedge$
  $now_s - first(t\text{-}out_s) \geq 2 \times \mu$
Eff: $mode_s :=$ closed

$crash_s$
Eff: if $mode_s \neq$ closed then
    $mode_s :=$ rec

$recover_s$
Pre: $mode_s =$ rec
Eff: $cache\_cc := \infty$
  $mode_s :=$ closed

Figure 9-9: *The first pair of steps is the basic message sending step for the server and the corresponding step to receive the segment for the client. Also the time-out, crash, and recovery steps*

receive-seg$_{sc}$(cc_rcvd, sn$_s$, ack$_s$, msg$_s$, FIN)	*send-seg$_{sc}$(cc_rcvd, sn$_s$, ack$_s$, msg$_s$, FIN)*
Eff: if ($mode_s \in$ {closed, syn-sent}) then {	Pre: $(now_s - time\text{-}sent_s \geq RTO) \wedge$
$send\text{-}rst_s :=$ true	$send\text{-}fin_s \wedge \neg push\text{-}data_s \wedge$
$rst\text{-}seq_s := ack_s$	$mode_s \in$ {fin-wait-1, last-ack
}	closing}
else if $mode_c \notin$ {rec, reset} $\wedge$	Eff: $time\text{-}sent_s := now_s$
$(cc\_send_c = cc\_rcvd_s)$ then {	
$send\text{-}ack_c :=$ true	
if $sn_s = ack_c \vee sn_s = ack_c + 1$ then {	
$push\text{-}data_c :=$ true	
if $mode_c =$ estb then	
$mode_c :=$ close-wait	
else if $mode_c =$ fin-wait-1 then	
$mode_c :=$ closing	
else if $mode_c =$ fin-wait-2 then	
$mode_c :=$ timed-wait	
if $sn_s = ack_c + 1$ then	
$rcv\text{-}buf_c := rcv\text{-}buf_c \cdot msg_s$	
$ack_c := sn_s + 1$	
$time\text{-}sent_c := 0$	
if $ack_s = sn_c + 1$ then {	
if $mode_c =$ closing then	
$mode_c :=$ timed-wait	
$msg_c :=$ null	
$ready\text{-}to\text{-}send_c :=$ false	
$send\text{-}fin_c :=$ false	
if $send\text{-}buf_c \neq \epsilon$ then	
$send\text{-}ack_c :=$ false	
}	
}	
}	
send-seg$_{cs}$(RST, ack$_c$, rst-seq$_c$)	*receive-seg$_{cs}$(RST, ack$_c$, rst-seq$_c$)*
Pre: $mode_c \in$ {closed, syn-sent}	Eff: if $mode_s \neq$ rec $\wedge\ rst\text{-}seq_c = ack_s$ then
$\wedge send\text{-}rst_c =$ true	$mode_s :=$ reset
Eff: $send\text{-}rst_c :=$ false	
receive-seg$_{sc}$(RST, ack$_s$, rst-seq$_s$)	*send-seg$_{sc}$(RST, ack$_s$, rst-seq$_s$)*
Eff: if $mode_c \neq$ rec $\wedge\ rst\text{-}seq_s = ack_c \vee$	Pre: $mode_s \in$ {closed, listen, syn-rcvd} $\wedge$
$(rst\text{-}seq_s = 0 \wedge ack_s = sn_c + 1)$	$send\text{-}rst_s =$ true
then $mode_c :=$ reset	Eff: $send\text{-}rst_s :=$ false
shut-down$_c$	*shut-down$_s$*
Pre: $mode_c =$ reset	Pre: $mode_s =$ reset
Eff: $mode_c :=$ closed	Eff: $mode_s :=$ closed

Figure 9-10: *The basic FIN segment from the server is on the right, and the corresponding action to receive this segment at the client is on the left. Also the reset and shut down steps.*

9.2 The specification of $\mathcal{TTCP}$

As is the case with the specification of TCP, we compose the client and server automata with channel automata described in Chapter 5. We define $\mathcal{TTCP}'$ to be the parallel composition of these automata. That is,

$$\mathcal{TTCP}' \triangleq \mathcal{TTCP}_c \| \mathcal{TTCP}_s \| Ch_{cs}(\mathcal{P}) \| Ch_{sc}(\mathcal{P}).$$

The set $\mathcal{P}$ of possible packets of the channels is instantiated with the packets that $\mathcal{TTCP}_c$ and $\mathcal{TTCP}_s$ can send and receive. To match the user interface of specifications S and D, we want the *send-seg*$_{cs}(p)$ and *send-seg*$_{sc}(p)$ actions of $\mathcal{TTCP}_c$ and $\mathcal{TTCP}_s$ respectively and the *receive-seg*$_{cs}(p)$ and the *receive-seg*$_{sc}(p)$ actions of $Ch_{cs}(\mathcal{P})$ and $Ch_{sc}(\mathcal{P})$ respectively which are output actions in $\mathcal{TTCP}'$ to be internal actions. Thus, we use the action hiding operator defined in Chapter 3 to "hide" these actions. Let

$$\begin{aligned}
\mathcal{A}_{TT} \triangleq\ & \{\textit{receive-seg}_{sc}(\textit{SYN, cc_rcvd, sn}_s\textit{, ack}_s)\} \cup \{\textit{receive-seg}_{sc}(\textit{cc_rcvd, sn}_s\textit{, ack}_s\textit{, msg}_s)\} \cup \\
& \{\textit{receive-seg}_{sc}(\textit{SYN, cc_rcvd, sn}_s\textit{, ack}_s\textit{, msg}_s)\} \cup \\
& \{\textit{receive-seg}_{sc}(\textit{SYN, cc_rcvd, sn}_s\textit{, ack}_s\textit{, msg}_s\textit{, FIN})\} \cup \\
& \{\textit{receive-seg}_{sc}(\textit{cc_rcvd, sn}_s\textit{, ack}_s\textit{, msg}_s\textit{, FIN})\} \cup \\
& \{\textit{receive-seg}_{sc}(\textit{RST, ack}_s\textit{, rst-seq}_s)\} \cup \{\textit{send-seg}_{cs}(\textit{RST, ack}_c\textit{, rst-seq}_c)\} \cup \\
& \{\textit{send-seg}_{cs}(\textit{SYN, cc_send, sn}_c\textit{, msg}_c)\} \cup \\
& \{\textit{send-seg}_{cs}(\textit{SYN, cc_send, sn}_c\textit{, msg}_c\textit{, FIN})\} \cup \\
& \{\textit{send-seg}_{cs}(\textit{cc_send, sn}_c\textit{, ack}_c\textit{, msg}_c)\} \cup \{\textit{receive-seg}_{cs}(\textit{cc_send, sn}_c\textit{, ack}_c\textit{, msg}_c)\} \cup \\
& \{\textit{send-seg}_{cs}(\textit{cc_send, sn}_c\textit{, ack}_c\textit{, msg}_c\textit{, FIN})\} \cup \\
& \{\textit{receive-seg}_{cs}\ (\textit{SYN, cc_send, sn}_c\textit{, msg}_c)\} \cup \\
& \{\textit{receive-seg}_{cs}\ (\textit{SYN, cc_send, sn}_c\textit{, msg}_c\textit{, FIN})\} \cup \\
& \{\textit{receive-seg}_{cs}(\textit{cc_send, sn}_c\textit{, ack}_c\textit{, msg}_c\textit{, FIN})\} \cup \\
& \{\textit{receive-seg}_{cs}(\textit{RST, ack}_c\textit{, rst-seq}_c)\} \cup \{\textit{send-seg}_{sc}(\textit{SYN, cc_rcvd, sn}_s\textit{, ack}_s)\} \cup \\
& \{\textit{send-seg}_{sc}(\textit{SYN, cc_rcvd, sn}_s\textit{, ack}_s\textit{, msg}_s)\} \cup \{\textit{send-seg}_{sc}(\textit{sn}_s\textit{, ack}_s\textit{, msg}_s\textit{, FIN})\} \cup \\
& \{\textit{send-seg}_{sc}(\textit{SYN, cc_rcvd, sn}_s\textit{, ack}_s\textit{, msg}_s\textit{, FIN})\} \cup \\
& \{\textit{send-seg}_{sc}(\textit{sn}_s\textit{, ack}_s\textit{, msg}_s)\} \cup \{\textit{send-seg}_{sc}(\textit{RST, ack}_s\textit{, rst-seq}_s)\}
\end{aligned}$$

The general timed automaton for T/TCP, $\mathcal{TTCP}$ is defined as:

$$\mathcal{TTCP} \triangleq \mathcal{TTCP}' \backslash \mathcal{A}_{TT}.$$

This definition gives a GTA with the same set of input and output actions as S and D.

This completes our formal modeling of T/TCP. When we first started doing this work, we initially though the next step in the verification of T/TCP would be to show a simulation from the model for T/TCP, to the model for TCP. However, in trying to do that simulation we observed that T/TCP does not behave like TCP, and no such simulation exists. Shankar and Lee in an earlier work [33] discovered the same situation, but we were unaware of their work when we made the observation. In the next section we describe this situation.

9.3 T/TCP behaves differently

In this section we show that no simulation exists from $\mathcal{TTCP}$ to $\mathcal{TCP}$ by describing an execution where the external user sees different behavior between the protocols. This behavior of $\mathcal{TTCP}$ also violates our specification in that the same data is delivered twice. The duplicate delivery in T/TCP occurs because the TAO mechanism bypasses the three-way handshake protocol in an effort to achieve efficient transactions. We first describe informally the situation where T/TCP behaves differently from TCP by delivering the same message twice, and then give the execution fragment of $\mathcal{TTCP}$ that cannot be simulated by any sequence of $\mathcal{TCP}$.

9.3.1 Duplicate delivery in T/TCP

The situation where the TAO mechanism cause T/TCP to deliver the same message twice is as follows. The T/TCP client gets a message to send from its user. When it gets this message all the persistent variables have values which allow it to sent a SYN segment for TCP accelerate open (TAO). The client sends the SYN segment and when the server receives it, the TAO test is successful. The server accepts the data and passes it to the user. Before the server can send a response a crash occurs. After the server recovers it reopens, and receives a retransmission of the segment it received before the crash. Since the crash might have caused the server to lose its persistent variables, it initiates a three-way handshake, by sending the second segment of the three-way handshake protocol to the client. When the client receives this segment, it cannot tell that the server accepted a previous copy of

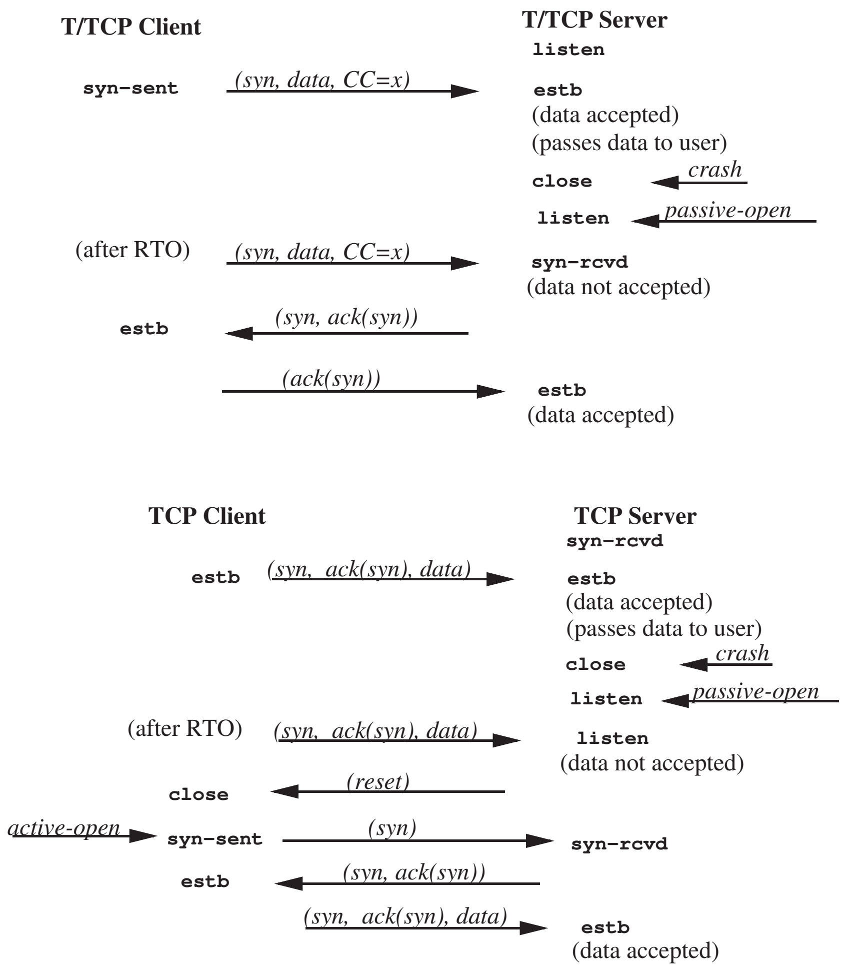

Figure 9-11: *The top figure shows T/TCP in the situation data is retransmitted after a crash and the bottom figure shows TCP in the same situation.*

the segment, and that the response is to a duplicate. Therefore, it sends the final segment of the three-way handshake. When the server receives this segment, it can accepted the data in the first segment and pass it to its user. This second delivery of the data means that the server delivers old duplicate data from the previous incarnation, which violates our specifications.

The sequence of actions in $\mathcal{TTCP}$ that gives the behavior described above is as follows. The client with $s.mode_c =$ `closed` gets the *send-msg*$_c$*(true, data1, false)* input. The client then sends a SYN segment for TAO. The server, also in mode `closed` opens via the *passive-open* input. Next the client performs the *prepare-msg*$_c$ action followed by the *send-seg*$_{cs}$*(SYN, cc_send, sn*$_c$*, data1)* action. The server gets this segment via the *receive-*

seg_{cs}*(SYN, cc_send,* sn_c*, data1)* action. For this execution the server has $s'.cache\_cc < s'.cc\_send$, so *data1* is concatenated to $s'.rcv\text{-}buf_s$. Next the server passes the data to the user with the $receive\text{-}msg_s$*(data1)* action. Immediately after that action, the server receives a $crash_s$ input. After a period of time the server recovers by issuing the $recover_s$ action. While the server is in recovery mode, the client repeatedly retransmits the *(SYN, cc_send,* sn_c*, data1)* segment.

After the server recovers, it receives the *passive-open* input from the user again and goes to mode `listen`. It next receives one of the retransmitted *(SYN, cc_send,* sn_c*, data1)* segments. Since there was a crash, *cache_cc* is not defined, so the TAO test fails. Therefore, the server assigns *data1* to *temp-data* and initiates the three-way handshake with the *send-*seg_{sc}*(SYN, cc_rcvd,* sn_s*,* ack_s*)* action. The client receives the segment with the *receive-*seg_{sc}*(SYN,* sn_s*,* ack_s*)* action and responds with the *send-*seg_{cs}*(*sn_c*,* ack_c*, data1)* action. The server receives the segment with the *receive-*seg_{cs}*(*sn_c*,* ack_c*, data1)* action. Since *temp-data* has the value `null`, *data1* is concatenated to $rcv\text{-}buf_s$. The server then performs the output action $receive\text{-}msg_s$*(data1)* again. Thus, the same data is delivered twice. We claim that this behavior is not allowed by TCP or by the specifications S or D.

The *trace* of the execution sequence described above is: $send\text{-}msg_c$*(true, data1, false)*, *passive-open*, $receive\text{-}msg_s$*(data1)*, $crash_s$, $recover_s$, *passive-open*, $receive\text{-}msg_s$*(data1)*.

9.3.2 No duplicate delivery in TCP

The trace just presented in the previous section, where *data1* is delivered twice, is not possible in specifications S or D. Thus, by Theorem 7.1 it is not possible in TCP. We know the trace is not possible in specifications S or D because after the server crashes in the specifications, all the data associated with the queue from which the server was receiving data is deleted and the queue becomes `dead`. Therefore, data can no longer be received from that queue. In order for the server to receive new data after it recovers and re-opens, the client must add data to a new queue and an association must be formed with the new id of the server and a new id chosen by the client. However, the client can only choose a new id, if it closes and then receives the $send\text{-}msg_c$*(open,m,close)* input from the user. This open input means the trace is different.

9.4 The next step

Since we now know that T/TCP does not implement TCP, there are two possible ways we could proceed with our work. The first way is to say that T/TCP is wrong and leave it at that or try to fix it. The second way to is to try to get a better understanding of the behavior of T/TCP. When we discussed the different behavior of T/TCP with the designers, they did not think this behavior was necessarily wrong. They though for some applications it might not matter that after a crash there is duplicate delivery. Therefore, we decided to proceed with the second option. To get a better understanding of the behavior of T/TCP, we decided to write a specification that captures the behavior of T/TCP and then show that the protocol satisfies this specification. The next chapter contains this work.

Chapter 10

Verification of T/TCP

Initially when we set out to verify T/TCP we thought we could show that it implements TCP, and since we prove TCP implements our specification of reliable transport level protocols (Chapters 6 and 8), by the transitivity of trace inclusion, we would have a proof that T/TCP also implements our specification. However, as we showed in Chapter 9, the user-visible behavior of T/TCP is different from that of TCP. One way to look at the fact that T/TCP exhibits different behavior from TCP and does not satisfy our specification is to say that it is wrong and should be corrected. Another way to look at this fact is to say that it is different, but not necessarily wrong for the type of applications the designers have in mind. In fact the designers of T/TCP and other network protocol designers that we have spoken to do not seem to think that this behavior of T/TCP is necessarily wrong. One argument they present as to why this behavior may not be wrong is that whatever effect receiving the message may have had on the user is probably lost after the crash, so receiving a second copy of the message is not exactly like receiving a duplicate. Therefore, in order to clearly understand the different behavior of T/TCP, we formulate a weaker specification that captures this behavior.

10.1 Weaker specifications

In specification S the same data is not allowed to be delivered twice under any circumstances. As we showed in the Chapter 9, in T/TCP a crashed host may reopen and receive duplicate data from the sender. Thus, we need to change specification S, so that it allows this behavior. However, we still want to maintain the other correctness properties of specification S — data is delivered in order; data is delivered without loss, except in the case of crashes; and data from different incarnations are separated. We also want to allow duplicate delivery only after crashes and aborts.

10.1.1 The primary changes

We make two primary changes to specification S to get a weaker version. The first of these changes in the weaker version of the specification is that we relax the rules for starting new incarnations. More specifically, we allow a host to continue with the same incarnation it had before a crash or an abort when it reopens after the crash or abort. The way we allow this to happen in the weaker version of specification S is by allowing the host to choose the same id it had before the crash or abort when it reopens after either event. However, if a host chooses a different id the first time it reopens after the crash or abort, then it cannot again choose the id it had at the time of the crash or abort. That is, if the host chooses to start a new incarnation immediately after the crash or abort, it cannot later choose to continue the previous incarnation.

Recall that in specification S every time a host re-opens it chooses a new id. Incarnations in specification S are identified by unique pairings of these id's. By unique pairings we mean an id is only allowed to be paired with a unique id from the other host. We call these pairs *associations*. Hosts are only allowed to accept data from a sender whose id is paired with the id of the receiving host. Thus, in specification S if the receiving host crashes, when it reopens it cannot accept data from the sending host unless the sender also closes and they both choose new id's that are paired together. In the weaker version of specification S, the pairings of id's are still unique, and a host can still only accept data from a sender whose id is paired with the id of the receiving host. However, because a host can choose the same id it had before a crash when it reopens after the crash, it can still accept data from the sending host even when the sending host does not close and reopen.

The second conceptual change is to allow duplicate delivery of data. We do not want to allow a piece of data to be delivered more than once in any arbitrary situation. We want to allow this to happen only on the last piece of data delivered for an incarnation, and only if the receiving host crashed or aborted after the delivery.

To incorporate these two ideas into the new specification, we need variables to keep track of the id a host has at the time of a crash or abort, so that it may be reused after the crash or abort, and we need to store the last piece of data delivered to a user.

10.1.2 Secondary changes

In specification S when a host crashes or aborts, the queue from which it is receiving data is emptied and the status is assigned `dead`. We do this because we know that in TCP after a crash or reset a host cannot receive any more data from the incarnation it was a part of before the crash or reset. Since data in each abstract queue represents data from a particular incarnation, killing and emptying the queue in specification S explicitly indicates that no more data from that incarnation should be received in the concrete implementations.

Since in T/TCP after a host crashes and reopens it may still receive data from the incarnation that it was receiving data from before the crash, in the new specification that captures the behaviors of T/TCP we cannot just kill and empty the abstract queues when the receiving hosts crash. Therefore, in our new specification we need to make some secondary changes to allow a host, after it reopens from a crash, to receive data from the same queue it received data from before the crash. The changes are as follows:

1. Queues are not emptied and killed when the receiving host crashes or aborts.
2. After crashes or aborts data may be lost from the front of queues.
3. Queues that are killed when the receiving host crashes or aborts in specification S, but are not killed in the weaker version of the specification, are now killed when the sender closes if data can no longer be received from these queues and they are still `live`. If data may still be received from these queues when the sender closes, they are killed by the receiving host when it is determined that data can no longer be received from these queues.

The reason why the weaker specification must now allow data to be lost from the front of queues after crashes or aborts is that in low level protocols when a receiving host crashes, it may lose its receive buffer. The receive buffer in a low level protocol corresponds to data from the front of an abstract queue. In specification S, queues that represent data going in one direction for an incarnation are killed when it is determined that no more data can be delivered from that queue. In the weaker specification these queues are killed when it is determined that no more data can be added to these queues and no more data can be received from these queues. If data can be received or added, these queues are not killed in the weaker specification.

In specification S, even if we did not kill and empty the queue when the receiving host crashed, the receiving host still could not receive any more data from that queue. The reason is that in specification S whenever a host crashes or aborts it must close and reopen before it can receive data again. When it reopens it must choose an new id, and the new id cannot be paired with the id of the queue that it was receiving data from before the crash because that id is already paired with the id the receiving host had before the crash. Since a host can only receive data from a queue indexed by an id that is paired with its current id, when the host reopens after the crash or abort, it cannot receive data from that queue. The reason why we chose to empty and kill the queue in specification S is that it makes the property more explicit, and it makes our refinement mapping for the simulation proof simpler.

When we make all these changes to S we get a weaker specification. The specification is weaker in that it allows more behaviors that specification S. We called this weaker specification for reliable TCP/IP transport level protocols $\mathcal{WS}$.

10.1.3 States and start states of $\mathcal{WS}$

The states and start states of $\mathcal{WS}$ remains the same as the state and start states of S except for the addition of four new variables. These variables record the crash id's of each host, and keeps track of the last messages passed to the users. We elaborate on these new variables in the table below.

Variable	Type	Initially	Description
$crash\text{-}id_c$	$CID \cup \{\texttt{nil}\}$	`nil`	The id_c value the client has at the last $crash_c$ or $abort_c$ event. If the client closes normally, it is set to `nil`.
$crash\text{-}id_s$	$SID \cup \{\texttt{nil}\}$	`nil`	Symmetric to $crash\text{-}id_c$.
$last\text{-}msg_c$	$Msg \cup \{\texttt{null}\}$	`null`	The most recent message passed to the user by the client. If there is a crash or an abort this message may be re-delivered.
$last\text{-}msg_s$	$Msg \cup \{\texttt{null}\}$	`null`	Symmetric to $last\text{-}msg_c$.

The derived variables $live\text{-}q_{cs}$ and $live\text{-}q_{sc}$ are defined for $\mathcal{WS}$ exactly as they are defined for S in Chapter 4.

The action signature for $\mathcal{WS}$ remains the same as for S presented in Section 4.1.2, except for the *lose* actions, which in $\mathcal{WS}$ have the form $lose_c(I,J,i)$ and $lose_s(I,J,i)$.

10.1.4 The steps

The steps of $\mathcal{WS}$ are shown below in Figures 10-1, 10-2, and 10-3. The effect of the $send\text{-}msg_c(open,\ m,\ close)$ action in $\mathcal{WS}$ changes to reflect that fact that $crash\text{-}id_c$ might be reused, but if it is not used, in the incarnation immediately after the crash or abort, it cannot be reused, so it is added back to the set of used client id's. Additionally, when $crash\text{-}id_c$ is not reused when the client reopens, if it is part of an association pair, but the server no longer has the id which $crash\text{-}id_c$ is paired with, then the queue indexed by the id that $crash\text{-}id_c$ is paired with is emptied and killed. The queue is killed because the server is no longer adding data to that queue, and since the current id of the client is not the id paired with that queue, the client can no longer receive data from it. The variable $crash\text{-}id_c$ is also assigned to `nil` in this step when the client opens whether it was reused or not, because once the client reopens, there is not longer a crash id until the next crash or abort. In the effect clause of the *choose-server-id(j)* symmetric assignments are made.

The preconditon of the *make-assoc(i,j)* action changes to allow $crash\text{-}id_c$ and $crash\text{-}id_s$, if they are not `nil`, to be part of association pairs, and not just the ids in the set of used id's. The reason for this change is that after a crash or an abort an id might get removed from the set of used id's, so it can be used again. However, we still want to allow that id to be able to form an association pair if it is not already part of a pair. When an id is reused, if it is already part of an association pair, no new association has to be formed for data to be sent and delivered on the queue indexed by that id. We still only allow id's to be paired with one other id.

The *receive-msg*$_c$*(m)* and *receive-msg*$_s$*(m)* steps in $\mathcal{WS}$ save a copy of the last message the client and server respectively passed to the user. This allows that message to be passed to the user again in the event that there is a crash.

The *set-nil*$_c(j)$ step is changed to include the assignment of $queue_{cs}(id_c)$ to ϵ and $q\text{-}stat_{cs}(id_c)$ to `dead` if the queue is `live` and $id_s \neq j \ \wedge j \in$ *used-id*$_s$. These assignments are made in this step because if the current id of the server is not the id that is paired with id_c and the id that is paired with id_c is used, then the server must have closed since id_c and j formed an association. If they server closes "normally," that is, with the *set-nil*$_s(j)$ action, then it kills $q\text{-}stat_{cs}(id_c)$. However, if the server closes because of a crash or an abort, it does not kill this queue, because the client might still add data to it after the crash or abort and when it reopens the server might still receive data from this queue. When the server crashes or aborts its id removed from the set of used id's, so the client will not kill the queue if the server as not reopened from a crash or an abort. The *set-nil*$_s(i)$ step is symmetric to *set-nil*$_c(j)$, and the effects of *reset-nil*$_c$ and *reset-nil*$_c$ actions remain unchanged from specification S.

The *crash*$_c$ and *abort*$_c$ steps assign id_c at the time of the crash or abort to the *crash-id*$_c$ set. In order to allow this id to be chosen by the client again, it is removed from the set of used client id's. Also if the queue from which the client is able to receive data before the crash or abort event is `live`, and *last-msg*$_c$ is not `null`, then *last-msg*$_c$ is added to the front of this queue to allow duplicate delivery of this message.

The *crash*$_c$ and *abort*$_c$ steps set rec_c and $abrt_c$ to `true` respectively. Either assignment enables the *lose*$_c$*(I,J,j)* action. This action changes to reflect the fact that after a crash at the client in $\mathcal{WS}$ not only may data be lost from the back of the queue on which the client has been putting data, but if there is a queue from which the client has been able to receive data that is still `live`, then data may be lost from the front of this queue. In this actions the set of indices J is an element of the set of prefixes of this queue. The definition of *prefixes* is analogous to the definition of *suffixes* in Chapter 4. That is, for any queue q

we define

$$prefixes(q) \triangleq \{\{i | 1 \leq i \leq j\} | 0 \leq j \leq |q|\}.$$

The fact that data may now be lost from the front of this queue is reflected in the precondition $((id_c, j) \in assoc \wedge qstat_{sc}(j) = \texttt{live} \wedge J \in prefixes(queue_{sc}(j))) \vee (\neg((id_c, j) \in assoc \wedge q\text{-}stat_{sc}(j) = \texttt{live}) \wedge j \in SID \wedge J = \emptyset)$. That is, if there is a live queue from which the client is able to receive data, elements may be lost from the front of that queue. Otherwise J is the empty set and j takes an arbitrary value. The $crash_s$ $abort_s$, and $lose_s(I,J,i)$ events are symmetric to their client side counterparts.

The $recover_c$ and $shut\text{-}down_c$ steps like $set\text{-}nil_c(j)$ sets $queue_{cs}(id_c)$ to empty, and its status to dead if the queue is live and $id_s \neq j \wedge j \in used\text{-}id_s$. The $recover_s$ and $shut\text{-}down_s$ actions are symmetric.

Invariants of $\mathcal{WS}$

Invariants 4.1, 4.2, and 4.3, defined for S in Section 4.1.2, also hold for $\mathcal{WS}$. The properties stated below are true of all reachable states of $\mathcal{WS}$.

Invariant 10.1

If $(h, j) \in u.assoc \wedge (i, j) \in u.assoc$ then $h = i$.

If $(i, j) \in u.assoc \wedge (i, k) \in u.assoc$ then $j = k$.

Proof: The proof is straightforward, by induction, from the description of the initial values of the variables of $\mathcal{WS}$ and of $steps(\mathcal{WS})$. ■

Invariant 10.2

$\forall\ i \in CID$, if $u.q\text{-}stat_{cs}(i) = \texttt{dead}$ then $u.queue_{cs}(i) = \epsilon$

$\forall\ j \in SID$, if $u.q\text{-}stat_{sc}(j) = \texttt{dead}$ then $u.queue_{sc}(j) = \epsilon$

Proof: The proof is straightforward, by induction, from the description of the initial values of the variables of $\mathcal{WS}$ and of $steps(\mathcal{WS})$. ■

Invariant 10.3

For any state u of $\mathcal{WS}$, $|u.live\text{-}q_{cs}|$ and $|u.live\text{-}q_{sc}|$ are both finite.

Proof: The proof is the same as the proof for Invariant 4.3. ■

The conjuction of the above invariants is itself an invariant which we call I_{WS}.

send-msg$_c$*(open, m, close)*
Eff: if $\neg(rec_c \vee abrt_c)$ then
  if $open \wedge id_c = \texttt{nil}$ then
    $id_c :\in\ CID \setminus used\text{-}id_c$
    $used\text{-}id_c := used\text{-}id_c \cup \{id_c\}$
    if $id_c \neq crash\text{-}id_c \wedge$
    $crash\text{-}id_c \neq \texttt{nil}$ then
      $used\text{-}id_c := used\text{-}id_c \cup \{crash\text{-}id_c\}$
      $\forall\, j$ s. t. $(crash\text{-}id_c, j) \in assoc$
      if $id_s \neq j \wedge$
      $q\text{-}stat_{sc}(j) = \texttt{live}$ then
        $q\text{-}stat_{sc}(j) := \epsilon$
        $q\text{-}stat_{sc}(j) := \texttt{dead}$
    $crash\text{-}id_c := \texttt{nil}$
    $last\text{-}msg_c := \texttt{null}$
    $mode_c := \texttt{active}$
    $q\text{-}stat_{cs}(id_c) := \texttt{live}$
  if $mode_c = \texttt{active} \wedge m \neq \texttt{null} \wedge$
  $q\text{-}stat_{cs}(id_c) = \texttt{live}$ then
    $queue_{cs}(id_c) := queue_{cs}(id_c) \cdot m$
  if $close$ then $mode_c := \texttt{inactive}$

make-assoc(i,j)
Pre: $i \neq \texttt{nil} \ \wedge\ i \in used\text{-}id_c\ \cup \{crash\text{-}id_c\} \wedge$
  $j \neq \texttt{nil} \wedge\ j \in used\text{-}id_s\ \cup \{crash\text{-}id_s\} \wedge$
  $\forall\, k\ (i,k) \notin assoc\ \wedge \forall\, l\ (l,j)\ \notin assoc$
Eff: $assoc := assoc \cup \{(i,j)\}$

receive-msg$_c$*(m)*
Pre: $\neg(rec_c \vee abrt_c) \wedge$
  $q\text{-}stat_{sc}(j) = \texttt{live}\ \wedge\ (id_c, j) \in assoc \wedge$
  $queue_{sc}(j) \neq\ \epsilon \wedge head(queue_{sc}(j)) = m$
Eff: $last\text{-}msg_c := head(queue_{sc}(j))$
  $queue_{sc}(j) := tail(queue_{sc}(j))$

set-nil$_c(j)$
Pre: $\neg(rec_c \vee abrt_c) \wedge$
  $id_c \neq \texttt{nil}\ \wedge mode_c = \texttt{inactive} \wedge$
  $(id_c, j) \in assoc\ \wedge queue_{sc}(j) = \epsilon \wedge$
  $(mode_s = \texttt{inactive}\ \vee id_s \neq j)$
Eff: if $q\text{-}stat_{cs}(id_c) = \texttt{live} \wedge$
  $id_s \neq j\ \wedge\ j \in used\text{-}id_s$ then
    $queue_{cs}(id_c) := \epsilon$
    $q\text{-}stat_{cs}(id_c) := \texttt{dead}$
  $id_c := \texttt{nil}$
  $q\text{-}stat_{sc}(j) := \texttt{dead}$

passive-open
Eff: if $\neg(rec_s \vee abrt_s)$ then
  if $id_s = \texttt{nil}$ then
    $choose\text{-}sid := \texttt{true}$
    $mode_s := \texttt{active}$
    $last\text{-}msg_s := \texttt{null}$

choose-server-id(j)
Pre: $choose\text{-}sid = \texttt{true} \wedge j \in SID\ \setminus used\text{-}id_s$
Eff: $choose\text{-}sid := \texttt{false}$
  $id_s := j$
  $used\text{-}id_s := used\text{-}id_s \cup \{j\}$
  if $id_s \neq crash\text{-}id_s \wedge crash\text{-}id_s \neq \texttt{nil}$ then
    $used\text{-}id_s := used\text{-}id_s \cup \{crash\text{-}id_s\}$
    $\forall\, i$ s. t. $(i, crash\text{-}id_s) \in assoc$
    if $id_c \neq i\ \wedge\ q\text{-}stat_{cs}(i) = \texttt{live}$ then
      $q\text{-}stat_{cs}(i) := \epsilon$
      $q\text{-}stat_{cs}(i) := \texttt{dead}$
  $crash\text{-}id_s := \texttt{nil}$
  $q\text{-}stat_{sc}(j) := \texttt{live}$

send-msg$_s$*(m, close)*
Eff: if $\neg(rec_s \vee abrt_s)$ then
  if $mode_s = \texttt{active} \wedge m\ \neq \texttt{null} \wedge$
  $q\text{-}stat_{sc}(id_s) = \texttt{live}$ then
    $queue_{sc}(id_s) := queue_{sc}(id_s) \cdot m$
  if $close$ then $mode_s := \texttt{inactive}$

receive-msg$_s$*(m)*
Pre: $\neg(rec_s \vee abrt_s) \wedge$
  $q\text{-}stat_{cs}(i) = \texttt{live} \wedge (i, id_s) \in assoc \wedge$
  $queue_{cs}(i) \neq \epsilon \wedge head(queue_{cs}(i)) = m$
Eff: $last\text{-}msg_s := head(queue_{cs}(i))$
  $queue_{cs}(i) := tail(queue_{cs}(i))$

set-nil$_s(i)$
Pre: $\neg(rec_s \vee abrt_s) \wedge$
  $id_s \neq \texttt{nil}\ \wedge mode_s = \texttt{inactive} \wedge$
  $(i, id_s) \in assoc\ \wedge queue_{cs}(i) = \epsilon \wedge$
  $(mode_c = \texttt{inactive}\ \vee id_c \neq i)$
Eff: if $q\text{-}stat_{sc}(id_s) = \texttt{live} \wedge$
  $id_c \neq i\ \wedge\ i \in ucid$ then
    $queue_{sc}(id_s) := \epsilon$
    $q\text{-}stat_{sc}(id_s) := \texttt{dead}$
  $id_s := \texttt{nil}$
  $q\text{-}stat_{cs}(i) := \texttt{dead}$

Figure 10-1: *Steps of the specification $\mathcal{WS}$.*

reset-nil$_c$
Pre: $\neg(rec_c \vee abrt_c) \wedge$
$id_c \neq$ `nil` $\wedge$ $mode_c =$ `inactive` $\wedge$
$\forall\, j(id_c, j) \notin assoc \wedge queue_{cs}(id_c) = \epsilon$
Eff: $id_c :=$ `nil`
$q\text{-}stat_{cs}(id_c) :=$ `dead`

crash$_c$
Eff: if $id_c \neq$ `nil` then
$rec_c :=$ `true`
$crash\text{-}id_c := id_c$
$used\text{-}id_c := used\text{-}id_c \setminus id_c$
$\forall\, j$ s.t. $q\text{-}stat_{sc}(j) =$ `live` $\wedge$
$(id_c, j) \in assoc$
if $last\text{-}msg_c \neq$ `null` then
$queue_{sc}(j) := last\text{-}msg_c \cdot queue_{sc}(j)$

abort$_c$
Pre: $id_c \neq$ `nil`
Eff: $abrt_c :=$ `true`
$crash\text{-}id_c := id_c$
$used\text{-}id_c := used\text{-}id_c \setminus id_c$
$\forall\, j$ s.t. $q\text{-}stat_{sc}(j) =$ `live` $\wedge$
$(id_c, j) \in assoc$
if $last\text{-}msg_c \neq$ `null` then
$queue_{sc}(j) := last\text{-}msg_c \cdot queue_{sc}(j)$

lose$_c$*(I,J,j)*
Pre: $(rec_c \vee abrt_c) \wedge$
$(I \in suffixes(queue_{cs}(id_c))) \wedge$
$(((id_c, j) \in assoc \wedge q\text{-}stat_{sc}(j) =$ `live` $\wedge$
$J \in prefixes(queue_{sc}(j))) \vee$
$(\neg((id_c, j) \in assoc \wedge q\text{-}stat_{sc}(j) =$ `live`$)$
$\wedge\, j \in SID \wedge J = \emptyset))$
Eff: $queue_{cs}(id_c) := \text{delete}(queue_{cs}(id_c), I)$
if $(id_c, j) \in assoc \wedge$
$q\text{-}stat_{sc}(j) =$ `live` then
$queue_{sc}(j) := \text{delete}(queue_{sc}(j), J)$

reset-nil$_s$
Pre: $\neg(rec_s \vee abrt_s) \wedge$
$id_s \neq$ `nil` $\wedge$ $mode_s =$ `inactive` $\wedge$
$\forall\, i(i, id_s) \notin assoc \wedge queue_{sc}(id_s) = \epsilon$
Eff: $id_s :=$ `nil`
$q\text{-}stat_{sc}(id_s) :=$ `dead`

crash$_s$
Eff: if $id_s \neq$ `nil` $\vee$ $mode_s =$ `active` then
$rec_s :=$ `true`
if $id_s \neq$ `nil` then
$crash\text{-}id_s := id_s$
$used\text{-}id_s := used\text{-}id_s \setminus id_s$
$\forall\, i$ s.t. $q\text{-}stat_{cs}(i) =$ `live` $\wedge$
$(i, id_s) \in assoc$
if $last\text{-}msg_s \neq$ `null` then
$queue_{cs}(i) := last\text{-}msg_s \cdot queue_{cs}(i)$

abort$_s$
Pre: $id_s \neq$ `nil` $\vee$ $mode_s =$ `active`
Eff: $abrt_s :=$ `true`
if $id_s \neq$ `nil` then
$crash\text{-}id_s := id_s$
$used\text{-}id_s := used\text{-}id_s \setminus id_s$
$\forall\, i$ s.t. $q\text{-}stat_{cs}(i) =$ `live` $\wedge$
$(i, id_s) \in assoc$
if $last\text{-}msg_s \neq$ `null` then
$queue_{cs}(i) := last\text{-}msg_s \cdot queue_{cs}(i)$

lose$_s$*(I,J,i)*
Pre: $(rec_s \vee abrt_s) \wedge$
$(I \in suffixes(queue_{sc}(id_s))) \wedge$
$(((i, id_s) \in assoc \wedge q\text{-}stat_{cs}(i) =$ `live`$) \wedge$
$J \in prefixes(queue_{cs}(i))) \vee$
$(\neg((i, id_s) \in assoc \wedge q\text{-}stat_{cs}(i) =$ `live`$)$
$\wedge\, i \in CID \wedge J = \emptyset))$
Eff: $queue_{sc}(id_s) := \text{delete}(queue_{sc}(id_s), I)$
if $(i, id_s) \in assoc \wedge$
$q\text{-}stat_{cs}(i) =$ `live` then
$queue_{cs}(i) := \text{delete}(queue_{cs}(i), J)$

Figure 10-2: *Other steps of the specification* $\mathcal{WS}$.

recover$_c$	*recover$_s$*
`Pre`: rec_c	`Pre`: rec_s
`Eff`: rec_c := `false`	`Eff`: rec_s := `false`
$mode_c$:= `inactive`	$mode_s$:= `inactive`
if $\forall\ j(id_c, j) \notin assoc \ \wedge queue_{cs}(id_c) = \epsilon$	if $\forall\ i(i, id_s) \notin assoc \wedge queue_{sc}(id_s) = \epsilon$
then optionally $q\text{-}stat_{cs}(id_c)$:= `dead`	then optionally $q\text{-}stat_{sc}(id_s)$:= `dead`
if $\exists\ j$ s.t. $(id_c, j) \in assoc\ \wedge$	if $\exists\ i$ s.t. $(i, id_s) \in assoc\ \wedge$
$q\text{-}stat_{cs}(id_c) =$ `live` $\wedge$	$q\text{-}stat_{sc}(id_s) =$ `live` $\wedge$
$id_s \neq j\ \wedge\ j \in used\text{-}id_s$ then	$id_c \neq i\ \wedge\ i \in used\text{-}id_c$ then
$queue_{cs}(id_c) := \epsilon$	$queue_{sc}(id_s) := \epsilon$
$q\text{-}stat_{cs}(id_c)$:= `dead`	$q\text{-}stat_{sc}(id_s)$:= `dead`
id_c := `nil`	id_s := `nil`
shut-down$_c$	*shut-down$_s$*
`Pre`: $abrt_c$	`Pre`: $abrt_s$
`Eff`: $abrt_c$:= `false`	`Eff`: $abrt_s$:= `false`
$mode_c$:= `inactive`	$mode_s$:= `inactive`
if $\forall\ j(id_c, j) \notin assoc \wedge queue_{cs}(id_c) = \epsilon$	if $\forall\ i(i, id_s) \notin assoc \wedge queue_{sc}(id_s) = \epsilon$
then optionally $q\text{-}stat_{cs}(id_c)$:= `dead`	then optionally $q\text{-}stat_{sc}(id_s)$:= `dead`
if $\exists\ j$ s.t. $(id_c, j) \in assoc\ \wedge$	if $\exists\ i$ s.t. $(i, id_s) \in assoc\ \wedge$
$q\text{-}stat_{cs}(id_c) =$ `live` $\wedge$	$q\text{-}stat_{sc}(id_s) =$ `live` $\wedge$
$id_s \neq j\ \wedge\ j \in used\text{-}id_s$ then	$id_c \neq i\ \wedge\ i \in used\text{-}id_c$ then
$queue_{cs}(id_c) := \epsilon$	$queue_{sc}(id_s) := \epsilon$
$q\text{-}stat_{cs}(id_c)$:= `dead`	$q\text{-}stat_{sc}(id_s)$:= `dead`
id_c := `nil`	id_s := `nil`

Figure 10-3: *The rest of the steps of the specification $\mathcal{WS}$.*

10.1.5 The weaker version of D

To get the weaker version of specification D, which we call $\mathcal{WD}$, we change $\mathcal{WS}$ in the same manner in which we changed S to get D. That is, messages are tagged when they are received in the *send-msg$_c$(open, m, close)* and *send-msg$_s$(m, close)* steps, and the tags are removed before the messages are passed to the users in the *receive-msg$_c$(m)* and *receive-msg$_s$(m)* steps. Also instead of enabling *lose* actions, crashes and aborts enable *mark* actions which mark the messages instead of deleting them. The marked messages maybe deleted in the *drop(I, J, k, l)* actions. These actions now reflect the fact that marked messages can be dropped from the front of a some queues as well as the back of queues as in D. Thus, the action *drop$_c$(I, J, k, l)* is enabled not only when there is a $queue_{cs}(k)$ which a suffix of marked messages, but also if there is $queue_{sc}(l)$ with a prefix of marked messages.

Depending on which condition causes the action to be enabled, the appropriate messages are deleted. The changed steps for $\mathcal{WD}$ are shown in Figures 10-4 and 10-5.

The derived variables $live\text{-}q_{cs}$, $live\text{-}q_{sc}$, and $\#\mathtt{ok}(q_D)$, where q_D is a queue in the set *(Msg* $\times$ *Flag)**, are defined for $\mathcal{WD}$ as they are defined for D. Similar to q_D, let q_S be a queue in the set *Msg**, that is, has the same type as queues in $\mathcal{WS}$.

send-msg$_c$*(open, m, close)*
Eff: if $\neg(rec_c \vee abrt_c)$ then
  if $open \wedge id_c = \mathtt{nil}$ then
    $id_c :\in \ CID \setminus used\text{-}id_c$
    $used\text{-}id_c := used\text{-}id_c \cup \{id_c\}$
    if $id_c \neq crash\text{-}id_c \wedge$
    $crash\text{-}id_c \neq \mathtt{nil}$ then
      $used\text{-}id_c := used\text{-}id_c \cup \{crash\text{-}id_c\}$
      $\forall\ j$ s. t. $(crash\text{-}id_c, j) \in assoc$
      if $id_s \neq j \wedge$
      $q\text{-}stat_{sc}(j) = \mathtt{live}$ then
        $q\text{-}stat_{sc}(j) := \epsilon$
        $q\text{-}stat_{sc}(j) := \mathtt{dead}$
    $crash\text{-}id_c := \mathtt{nil}$
    $last\text{-}msg_c := \mathtt{null}$
    $mode_c := \mathtt{active}$
    $q\text{-}stat_{cs}(id_c) := \mathtt{live}$
  if $mode_c = \mathtt{active} \wedge\ m \neq \mathtt{null} \wedge$
  $q\text{-}stat_{cs}(id_c) = \mathtt{live}$ then
    $queue_{cs}(id_c) := queue_{cs}(id_c) \cdot (m, \mathtt{ok})$
  if $close$ then $mode_c := \mathtt{inactive}$

receive-msg$_c$*(m)*
Pre: $\neg(rec_c \vee abrt_c) \wedge queue_{sc}(j) \neq \ \epsilon \wedge$
  $q\text{-}stat_{sc}(j) = \mathtt{live} \wedge (id_c, \mathrm{j}) \in assoc \wedge$
  $(\mathrm{head}(queue_{sc}(j))).\mathrm{msg} = m$
Eff: $last\text{-}msg_c := head(queue_{sc}(j))$
  $queue_{sc}(j) := \mathrm{tail}(queue_{sc}(j))$

send-msg$_s$*(m, close)*
Eff: if $\neg(rec_s \vee abrt_s)$ then
  if $mode_s = \mathtt{active} \wedge m\ \neq \mathtt{null} \wedge$
  $q\text{-}stat_{sc}(id_s) = \mathtt{live}$ then
    $queue_{sc}(id_s) := queue_{sc}(id_s) \cdot (m, \mathtt{ok})$
  if $close$ then $mode_s := \mathtt{inactive}$

receive-msg$_s$*(m)*
Pre: $\neg(rec_s \vee abrt_s) \wedge queue_{cs}(i) \neq \ \epsilon \wedge$
  $q\text{-}stat_{cs}(i) = \mathtt{live} \wedge (i, id_s) \in assoc \wedge$
  $(\mathrm{head}(queue_{cs}(i))).\mathrm{msg} = m$
Eff: $last\text{-}msg_s := \mathrm{head}(queue_{cs}(i))$
  $queue_{cs}(i) := \mathrm{tail}(queue_{cs}(i))$

Figure 10-4: *Steps of* $\mathcal{WD}$ *that differ from the steps of* $\mathcal{WS}$*.*

$mark_c(I, J, j)$
`Pre`: $(rec_c \lor abrt_c) \land$
$(I \in suffixes(queue_{cs}(id_c))) \land$
$(((id_c, j) \in assoc \land q\text{-}stat_{sc}(j) = \texttt{live} \land$
$J \in prefixes(queue_{sc}(j))) \lor$
$(\neg((id_c, j) \in assoc \land q\text{-}stat_{sc}(j) = \texttt{live})$
$\land\ j \in SID \land J = \emptyset))$
`Eff`: $queue_{cs}(id_c) := \text{mark}(queue_{cs}(id_c), I)$
if $(id_c, j) \in assoc \land q\text{-}stat_{sc}(j) = \texttt{live}$
then $queue_{sc}(j) := \text{mark}(queue_{sc}(j), J)$

$drop_c(I, J, k, l)$
`Pre`: $(q\text{-}stat_{cs}(k) = \texttt{live} \land$
$I \in suffixes(queue_{cs}(k)) \land$
$\forall\ i \in I\ queue_{cs}(k)[i].flag = \texttt{marked})$
$\lor\ (q\text{-}stat_{sc}(l) = \texttt{live} \land$
$J \in prefixes(queue_{sc}(l)) \land$
$\forall\ j \in J\ queue_{sc}(l)[j].flag = \texttt{marked})$
`Eff`: if $(q\text{-}stat_{cs}(k) = \texttt{live} \land,$
$I \in suffixes(queue_{cs}(k)) \land \forall\ i \in I$
$queue_{cs}(k)[i].flag = \texttt{marked})$ then
$queue_{cs}(k) := \text{delete}(queue_{cs}(k), I)$
if $(q\text{-}stat_{sc}(l) = \texttt{live} \land$
$J \in prefixes(queue_{sc}(l)) \land \forall\ j \in J,$
$queue_{sc}(l)[j].flag = \texttt{marked})$ then
$queue_{sc}(l) := \text{delete}(queue_{sc}(l), J)$

$mark_s(I, J, i)$
`Pre`: $(rec_s \lor abrt_s) \land$
$(I \in suffixes(queue_{sc}(id_s))) \land$
$(((i, id_s) \in assoc \land q\text{-}stat_{cs}(i) = \texttt{live}) \land$
$J \in prefixes(queue_{cs}(i)))\lor$
$(\neg((i, id_s) \in assoc \land q\text{-}stat_{cs}(i) = \texttt{live})$
$\land\ i \in CID \land J = \emptyset))$
`Eff`: $queue_{sc}(id_s) := \text{mark}(queue_{sc}(id_s), I)$
if $(i, id_s) \in assoc \land q\text{-}stat_{cs}(i) = \texttt{live}$
then $queue_{cs}(i) := \text{mark}(queue_{cs}(i), J)$

$drop_s(I, J, l, k)$
`Pre`: $(q\text{-}stat_{sc}(l) = \texttt{live} \land$
$I \in suffixes(queue_{sc}(l)) \land$
$\forall\ i \in I\ queue_{sc}(l)[i].\text{flag} = \texttt{marked})$
$\lor\ (q\text{-}stat_{cs}(k) = \texttt{live} \land$
$J \in prefixes(queue_{cs}(k))$
$\land \forall\ j \in J\ queue_{cs}(k)[j].flag = \texttt{marked})$
`Eff`: if $(q\text{-}stat_{sc}(l) = \texttt{live} \land$
$I \in suffixes(queue_{sc}(l)) \land \forall\ i \in I$
$queue_{sc}(l)[i].\text{flag} = \texttt{marked})$ then
$queue_{sc}(l) := \text{delete}(queue_{sc}(l), I)$
if $(q\text{-}stat_{cs}(k) = \texttt{live} \land$
$J \in prefixes(queue_{cs}(k)) \land \forall\ j \in J,$
$queue_{cs}(k)[j].flag = \texttt{marked})$ then
$queue_{cs}(k) := \text{delete}(queue_{cs}(k), J)$

Figure 10-5: *The other steps of $\mathcal{WD}$ that differ from the steps of $\mathcal{WS}$.*

10.1.6 The correctness of $\mathcal{WD}$

In this section we prove the correctness of $\mathcal{WD}$ with respect to $\mathcal{WS}$. We start by defining invariants on the states of $\mathcal{WD}$. The invariants are the same as Invariants 10.1 through 10.3. The properties stated below are true of all reachable states of $\mathcal{WD}$.

Invariant 10.4

If $(h, j) \in u.assoc \land (i, j) \in u.assoc$ then $h = i$.

If $(i, j) \in u.assoc \land (i, k) \in u.assoc$ then $j = k$.

Proof: The proof is straightforward, by induction, from the description of the initial values of the variables of $\mathcal{WD}$ and of *steps($\mathcal{WD}$)*. ∎

Invariant 10.5

$\forall\ i\ \in CID$, if $u.q\text{-}stat_{cs}(i) = \texttt{dead}$ then $u.queue_{cs}(i) = \epsilon$

$\forall\ j\ \in SID$, if $u.q\text{-}stat_{sc}(j) = \texttt{dead}$ then $u.queue_{sc}(j) = \epsilon$

Proof: The proof is the same as the proof for Invariant 10.2. ■

Invariant 10.6

For any state u of $\mathcal{WS}$, $|u.live\text{-}q_{cs}|$ and $|u.live\text{-}q_{sc}|$ are both finite.

Proof: The proof is the same as the proof for Invariant 4.3. ■

The conjuction of the above invariants is itself an invariant which we call I_{WD}.

The simulation

We prove the correctness of $\mathcal{WD}$ by showing an image finite backward simulation from $\mathcal{WD}$ to $\mathcal{WS}$. The proof is very similar to the one given in Chapter 4 for showing an image finite backward simulation from D to S. We will also use most of the definitions and preliminary lemmas from that proof.

We define B_{WDWS} over *states($\mathcal{WD}$)* $\times$ *states($\mathcal{WS}$)*. Definition 4.1 defines an explanation.

Definition 10.1 (Image-Finite Backward Simulation from $\mathcal{WD}$ to $\mathcal{WS}$)
If $s\ \in$ *states(D)* and $u\ \in$ *states(S)*, then define that $(s, u)\ \in B_{\text{WDWS}}$ if the following conditions hold:

1. $u.assoc = s.assoc$
2. $u.choose\text{-}sid = s.choose\text{-}sid$
3. $u.used\text{-}id_c = s.used\text{-}id_c$
   $u.used\text{-}id_s = s.used\text{-}id_s$
4. $u.rec_c = s.rec_c$
   $u.rec_s = s.rec_s$
5. $u.abrt_c = s.abrt_c$
   $u.abrt_s = s.abrt_s$
6. $u.id_c = s.id_c$
   $u.id_s = s.id_s$
7. $u.mode_c = s.mode_c$
   $u.mode_s = s.mode_s$

8. $u.crash\text{-}id_c = s.crash\text{-}id_c$
   $u.crash\text{-}id_s = s.crash\text{-}id_s$

9. $u.last\text{-}msg_c = s.last\text{-}msg_c$
   $u.last\text{-}msg_s = s.last\text{-}msg_s$

10. $\forall\ i\ \in CID\ u.q\text{-}stat_{cs}(i) = s.q\text{-}stat_{cs}(i)$
    $\forall\ j\ \in SID\ u.q\text{-}stat_{sc}(j) = s.q\text{-}stat_{sc}(j)$

11. $(\forall\ i\ \in CID)$ ($\exists$ explanation f_i from $u.queue_{cs}(i)$ to $s.queue_{cs}(i)$)
    $(\forall\ j\ \in SID)$ ($\exists$ explanation g_i from $u.queue_{sc}(j)$ to $s.queue_{sc}(j)$)

Each of the variables in $\mathcal{WS}$ other than the queues is equal to its counterpart in $\mathcal{WD}$. In the proof below when we write u.variables = s.variables we mean the eleven sets of equations of items one through nine in Definition 10.1.

Recall that in Chapter 4 we define *maxqueue* be a function of type: *(Msg* $\times$ *Flag)** $\rightarrow$ *Msg** such that for any q_D, *maxqueue(*q_D*)* is defined to be the queue q_S obtained by removing all flag components from q_D.

Lemma 10.1

Let $s\ \in$ states($\mathcal{WD}$). Then there exists a state $u\ \in$ states($\mathcal{WS}$) such that $(s, u)\ \in B_{\mathrm{WDWS}}$.

Proof: Let q_{S_i} = *maxqueue(s.queue*$_{cs}(i)$*)* $\forall\ i \in CID$, and $q^1_{S_j}$ = *maxqueue(s.queue*$_{sc}(j)$*)* $\forall\ j\ \in SID$. Then by Lemma 4.3 there exists an explanation from q_{S_i} to $s.queue_{cs}(i)$ and an explanation from $q^1_{S_j}$ to $s.queue_{sc}(j)$. Thus, if we have $u.queue_{cs}(i) = q_{S_i}$, $u.queue_{sc}(j) = q^1_{S_j}$, and for all the other variables u.variables = s.variables, this gives a state u such that $(s, u)\ \in B_{\mathrm{WDWS}}$. ■

Lemma 10.2

$\mathcal{WD} \leq_{iB} \mathcal{WS}$ via B_{WDWS} with respect to I_{WD} and I_{WS}.

Proof: We first show that B_{WDWS} is image-finite and then check the three conditions of Definition 3.10 which we call non-emptiness, base case, and inductive case respectively.

Let s be an arbitrary state of $\mathcal{WD}$. The proof that there are only finitely many states u of $\mathcal{WS}$ such that $(s, u) \in B_{\mathrm{WDWS}}$ is the same as the proof that for an arbitrary state s' of D, there exists only finitely many states u' of S such that $(s', u') \in B_{\mathrm{WDWS}}$ presented in Section 4.2.2, so we do not repeat it here.

Non-emptiness

Non-emptiness follows immediately from Lemma 10.1

Base Case

Let s_0 be the (unique) start state of $\mathcal{WD}$. Then if $(s_0, u) \in B_{\text{WDWS}}$, then u.variables = s.variables and $u.queue_{cs}(i) = u.queue_{sc}(j) = \epsilon$. Thus, u is the unique start state of $\mathcal{WS}$.

Inductive Case

The proof for the inductive case for this lemma is has many of the same steps as the proof for the inductive case for Lemma 4.5. We note the cases that are the same here, but do not repeat the arguments when this is the case.

Assume $(s, a, s') \in$ *steps*($\mathcal{WD}$) and let u' be an arbitrary state of $\mathcal{WS}$ such that $(s', u') \in B_{\text{WDWS}}$. Below we consider cases based on a and for each case we define a finite execution fragment α of S with *lstate*(α) = u', (s, *fstate*(α)) $\in B_{\text{WDWS}}$, and *trace*(α) = *trace*(a). In order to show (s, *fstate*(α)) $\in B_{\text{WDWS}}$, we need to show that the value of the state variables for state s and *fstate*(α) = u are related according to our definition of B_{WDWS}. As is the case for Lemma 4.5, the interesting aspect of showing $(s, u) \in B_{\text{WDWS}}$ is showing that we can find valid explanations from the queues in state u to the queues in state s.

a = *send-msg*$_c$*(open, m, close)*.

The proof for this case is the same as the proof for the same case of Lemma 4.5.

a = *passive-open, choose-server-id(j)*.

The proof for these cases is the same as the proof for the same cases of Lemma 4.5.

a = *send-msg*$_s$*(m, close)*.

The proof for this case is the same as the proof for the same case of Lemma 4.5.

a = *make-assoc(i,j)*.

The proof for this case differs from the proof for the same case of Lemma 4.5, because now the step with this action may change the queues. For this case α = *(u, make-assoc(i,j), u')*. We now define u such that $\alpha \in$ *steps(S)* and $(s, u) \in B_{\text{WDWS}}$. We let u.variables = s.variables. The only queues that change because of this step are $s.queue_{sc}(k)$ $\forall\ k \neq j$ s.t. $(i, k) \in$ *assoc* and $s.queue_{cs}(l)$ $\forall\ l \neq i$ s.t. $(l, j) \in$ *assoc*. For all other queues, the explanations from these queues in state u' to state s' are also explanations from state

u to state s. For the other queues that change in this step, $\forall\ k \neq j$ s.t. $(i,k) \in assoc$ let $u.queue_{sc}(k) = maxqueue(s.queue_{cs}(k))$, then by Lemma 4.3 the identity mapping from *dom(u.queue*$_{sc}(k)$*)* to *dom(s.queue*$_{sc}(k)$*)* is an explanation. Also let $u.queue_{cs}(l) = maxqueue(s.queue_{cs}(l)), \forall\ l \neq i$ s.t. $(l,j) \in assoc$, then by Lemma 4.3, the identity mapping is an explanation from $u.queue_{cs}(l)$ to $s.queue_{cs}(l)$.

$\underline{a = \textit{receive-msg}_c(m),\ \textit{receive-msg}_s(m)}$.

The proof for these cases is the same as the proof for the same cases of Lemma 4.5.

$\underline{a = \textit{reset-nil}_c,\ \textit{reset-nil}_s,\ \textit{recover}_c,\ \textit{recover}_s,\ \textit{shut-down}_c,\ \textit{shut-down}_s}$.

The proof for these cases is the same as the proof for the same cases of Lemma 4.5.

$\underline{a = \textit{set-nil}_c(j)}$.

For this case let $\alpha = (u, make - assoc(i,j), u')$. The proof for this case also changes, because in $\mathcal{WD}$ and $\mathcal{WS}$ the step now affects may now affect $s.queue_{cs}(s.id_c)$. Again u.variables $=$ s.variables and for all other queues, the explanations that exist from state u' to s' also hold from state u to state s. Let $u.queue_{cs}(u.id_c) = maxqueue(s.queue_{cs}(s.id_c))$, then by Lemma 4.3, the identity mapping from *dom(u.queue*$_{cs}(u.\mathrm{id}_c)$*)* to *dom(s.queue*$_{cs}(s.\mathrm{id}_c)$*)* is an explanation.

$\underline{a = \textit{set-nil}_s(i)}$.

The proof for this case is symmetric to the proof for $a = \textit{set-nil}_c(j)$.

$\underline{a = \textit{crash}_c}$.

We can define u such *(u, crash*$_c$*, u′)* $\in$ *steps(*$\mathcal{WS}$*)* and $(s,u)\ \in B_{\mathrm{WDWS}}$. For this step u.variables $=$ s.variables. This step only affects $queue_{sc}(j)$ if $s.\textit{q-stat}_{sc}(j) = \texttt{live}\ \wedge$ $(s.id_c, j) \in s.assoc$. For all other queues the explanations that exists from state u' to state s', also exists from state u to state s. Let f'_j be an explanation from $u'.queue_{sc}(j)$ to $s'.queue_{sc}(j)$. Then we can define f_j in the following way.

$$f_j = [i \mapsto (f'_j(i+1) - 1) | i \in dom(f'_j) \setminus maxindex(u'.queue_{sc}(j))]$$

Intuitively f_j relates the same elements in $u.queue_{sc}(j)$ and $s.queue_{sc}(j)$ that were related by f'_j in $u'.queue_{sc}(j)$ and $s'.queue_{sc}(j)$ (these elements all have their indices decreased by one because of the elements removed from the head of the queues). It is easy to see that f_j is an explanation.

$a = crash_s$.

This case is symmetric to the case for $crash_c$.

$a = abort_c$.

For this case we define u such that *(u, $abort_c$, u')* $\in$ *steps($\mathcal{WS}$)* and $(s, u) \in B_{\mathrm{WDWS}}$. Clearly the traces are the same. The proof for this case is the same as the proof for $a = crash_c$ because the effects of these actions are essentially the same.

$a = abort_s$.

This case is symmetric to the case for $abort_c$.

$a = mark_c(I, J, j)$.

In this case we can define u, I', and J' such that *(u, $lose_c(I', J', j)$, u')* $\in$ *steps($\mathcal{WS}$)* and $(s, u) \in B_{\mathrm{WDWS}}$. Clearly *trace($\alpha$)* $=$ *trace(a)*. Let u.variables $=$ s.variables. The action $mark_c(I, J, j)$ only affects $s.queue_{cs}(s.id_c)$ and $s.queue_{sc}(j)$ where $(((s.id_c, j) \in s.assoc \wedge s.q\text{-}stat_{sc}(j) = \texttt{live}) \Rightarrow J \in prefixes(s.queue_{sc}(j))$. Similarly the action $lose_c(I', J', j)$ only affects $u.queue_{cs}(u.id_c)$ and $u.queue_{sc}(j)$ where $(((u.id_c, j) \in u.assoc \wedge u.q\text{-}stat_{sc}(j) = \texttt{live}) \Rightarrow J \in prefixes(u.queue_{sc}(j))$. Therefore, for all other queues the explanations that exists from state u' to state s', also exists from state u to state s. Therefore, we need to construct explanations from $u.queue_{cs}(u.id_c)$ to $s.queue_{cs}(s.id_c)$ and from $u.queue_{sc}(j)$ to $s.queue_{sc}(j)$. Let $u.queue_{cs}(u.id_c) = maxqueue(s.queue_{cs}(s.\mathrm{id}_c))$ and $u.queue_{sc}(j) = maxqueue(s.queue_{sc}(j))$; then by

Lemma 4.3, the identity mapping is an explanation from $u.queue_{cs}(u.id_c)$ to $s.queue_{cs}(s.id_c)$ and from $u.queue_{sc}(j)$ to $s.queue_{sc}(j)$.

We now need to show that $lose_c(I', J', j)$ is enabled from state u in $\mathcal{WS}$. Since u.variables $=$ s.variables and $mark_c(I, J, j)$ is enabled in s, we know $s.rec_c = u.rec_c = \texttt{true}$, and that $I \in suffixes(s.queue_{cs}(s.id_c)$ and $(((s.id_c, j) \in s.assoc \wedge s.q\text{-}stat_{sc}(j) = \texttt{live}) \Rightarrow J \in prefixes(s.queue_{sc}(j))$. To define an appropriate I' and J' we first observe that $maxqueue(s.queue_{cs}(s.id_c)) = maxqueue(s'.queue_{cs}(s'.id_c))$ and $maxqueue(s.queue_{sc}(j)) = maxqueue(s'.queue_{sc}(j))$. Since $u.queue_{cs}(u.id_c) =$

$maxqueue(s.queue_{cs}(s.id_c))$ and $u.queue_{sc}(j) = maxqueue(s.queue_{sc}(j))$, it is easy to see we can obtain $u'.queue_{cs}(u'.id_c)$ from $u.queue_{cs}(u.id_c)$ and $u'.queue_{sc}(j)$ from $u.queue_{sc}(j)$ by deleting some (possibly zero) elements that are in $suffixes(u.queue_{cs}(u.id_c))$ and $prefixes(s.queue_{sc}(j))$ respectively. Thus, I is an appropriate I' and J is an appropriate J'. That is, $I' = I$ and $J' = J$.

$\underline{a = mark_s(I, J, j)}$.

This action is symmetric to the previous case.

$\underline{a = drop_c(I, J, k, l)}$.

The corresponding action in $\mathcal{WS}$ is the empty step, i.e., $(s, u') \in B_{\text{WDWS}}$. Since $drop_c(I, J, k, l)$ is internal the empty step has the right trace. This action only affects $s.queue_{cs}(k)$ and $s.queue_{sc}(l)$, so we only need explanations from $u.queue_{cs}(k)$ to $s.queue_{cs}(k)$ and from $u.queue_{sc}(l)$ to $s.queue_{sc}(l)$. In the proof of Lemma 4.5 for the case of the $drop_c(I, k)$ action, it is shown that if we let f'_k be an arbitrary explanation form $u'.queue_{cs}(k)$ to $s'.queue_{cs}(k)$ (we know one exists because $(s', u') \in B_{\text{WDWS}}$), and let h be the unique bijective, strictly increasing mapping from $dom(s'.queue_{cs}(k))$ to $dom(s.queue_{cs}(k)) \setminus I$, then $f_k = h \circ f'_k$ is a valid explanation from $u.queue_{cs}(k)$ to $s.queue_{cs}(k)$.

Now we only need to show an explanation from $u.queue_{sc}(l)$ to $s.queue_{sc}(l)$. The same technique used for finding an explanation from $u.queue_{cs}(k)$ to $s.queue_{cs}(k)$ can be used here. Let f'_l be an arbitrary explanation form $u'.queue_{sc}(l)$ to $s'.queue_{sc}(l)$ (we know one exists because $(s', u') \in B_{\text{WDWS}}$). J contains the indices of the elements of $s.queue_{sc}(l)$ that may be deleted in the $drop_c(I, J, k, l)$ step. Then $|dom(s'.queue_{sc}(l))| = |dom(s.queue_{sc}(l)) \setminus J|$. Now let g be the unique bijective, strictly increasing mapping from $dom(s'.queue_{sc}(l))$ to $dom(s.queue_{sc}(l)) \setminus J$. Informally, g maps indices of elements in $s'.queue_{sc}(l)$ to the indices the same elements had in $s.queue_{sc}(l)$. Define $f_l = g \circ f'_l$. The proof that f_l is a valid explanation is the same as the proof that f_k is a valid explanation.

$\underline{a = drop_s(I, J, l, k)}$.

This action is symmetric to the previous case.

This concludes the backward simulation proof. ■

Theorem 10.1

The traces of $\mathcal{WD}$ are a subset of the traces of $\mathcal{WS}$, that is, $\mathcal{WD} \sqsubseteq \mathcal{WS}$.

Proof: The proof follows directly from Lemma 10.2 and the soundness of backward simulations (Theorem 3.4). ■

10.2 $\mathcal{TTCP}$ with history variables

To verify that $\mathcal{TTCP}$ implements the weaker specification, we follow the same general set of steps used for the verification of TCP. That is, we want to show a simulation relation from the states of $\mathcal{TTCP}$ to the states of $\mathcal{WS}$. However, because of the non-determinism in the actual T/TCP protocol we use the intermediate weak Delayed Decision Specification $\mathcal{WD}$. Thus, we would like to show a refinement mapping from $\mathcal{TTCP}$ to $\mathcal{WD}$, but since $\mathcal{WD}$ is an untimed automaton and $\mathcal{TTCP}$ is a timed automaton we first need to apply the patient operator to $\mathcal{WD}$ to get the *patient($\mathcal{WD}$)* denoted as $\mathcal{WD}^p$. Before we can define a refinement mapping from the states of $\mathcal{TTCP}$ to the states of $\mathcal{WD}^p$ we also need to add some history variables to $\mathcal{TTCP}$. We call the resulting automaton *history T/TCP*, denoted as $\mathcal{TTCP}^h$. Most of the history variables are equivalent to variables with the same names in $\mathcal{WD}$. These history variables are id_c, id_s, *used-*id_c, *used-*id_s, *crash-*id_s, *last-*msg_c, *last-*msg_s, and *assoc*. In T/TCP the server echos back the value of the connection count to the client to verify that segments are from the current incarnation, so the connection count issued by the client is really the only value used to identify an incarnation in T/TCP. Thus, the history variable id_c is the *cc_send* value of the client when it opens, and the history variable id_s is the *cc_rcvd* value of the server, but only when the connection is established. That is, when the server knows that the [*cc_send*] value it receives on a segment is not an old duplicate. We also add history variables isn_s, which is the initial sequence number of the server for an incarnation. We need this history variable, because if the TAO test fails at the server, a three-way handshake is initiated, and the initial sequence number of the server is important in deterimining if the three-way handshake is valid. Related to the isn_s history variable is the *estb-cc* history variable. This variable is similar to the history variable *estb-pairs* of $\mathcal{TCP}^h$. It is the set of id_c values of the client paired with the isn_s values of the server after the second step of the three-way handshake. Thus, *estb-cc* records pairs that indicate the second leg of the three-way handshake as been successfully completed. We also have the history variable *sent-tao-cc* which records all the id_c values of client when it sends SYN segments with data. The history variable *choose-isn*$_s$ becomes true in the step that causes the server to choose an initial sequence number, and becomes false in any subsequent steps. The type and initial value of the history variables are shown in the table below.

Variable	**Type**	**S**	**Initially**	**Description**
id_c	$\mathbb{N} \cup$ `nil`		`nil`	The connection count each time the client opens.
id_s	$\mathbb{N} \cup$ `nil`		`nil`	The *cc_send* whenever a connection is successfully established at the server.
isn_s	$\mathbb{N} \cup$ `nil`		`nil`	The server side intial sequence number.
used-id$_c$	$2^{\mathbb{N}}$	$\surd$	$\emptyset$	The set of id's used by the client.
used-id$_s$	$2^{\mathbb{N}}$	$\surd$	$\emptyset$	The set id's used by the server.
crash-id$_c$	$\mathbb{N}$	$\surd$	$\emptyset$	The id_c value whenever the client crashes or resets.
crash-id$_s$	$\mathbb{N}$	$\surd$	$\emptyset$	Symmetric to *crash-id*$_c$.
last-msg$_c$	*Msg**		`null`	The last message passed to the user on the client side
last-msg$_s$	*Msg**		`null`	Symmetric to *last-msg*$_c$
assoc	$2^{(\mathbb{N}\times\mathbb{N})}$	$\surd$	$\emptyset$	A set of pairs of *id*'s for each incarnation of the connection.
estb-cc	$2^{\mathbb{N}\times\mathbb{N}}$	$\surd$	$\emptyset$	The set of pairs of id_c values the client has paired with the initial sequence number of the sever, whenever the gets to mode `estb` as a result of receiving the second segment in the three-way handshake.
choose-isn$_s$	Bool		`false`	A flag that is set to `true` when the server first chooses an ISN for an incarnation and set to `false` when the server sends a segment with this ISN.
sent-tao-cc	$2^{\mathbb{N}}$	$\surd$	`nil`	The set of connection count values of SYN segments that the client sends valid data.

10.2.1 Steps of $\mathcal{TTCP}^h$

The steps of $\mathcal{TTCP}^h$ that differs from $\mathcal{TTCP}$ are show In Figures 10-6, 10-7, and 10-8. As always we omit the assignments to the original variables of $\mathcal{TTCP}$ (again indicated by ...) but outline the if-then-else statements. The first addition is to the *send-msg*$_c$*(open, m, close)* step. When the client opens for an incarnation, id_c is assigned the new value of *cc_send*. In this step in specification $\mathcal{WD}$, several variables that have corresponding history variables in $\mathcal{TTCP}^h$ get assigned, so the corresponding assignments are made for $\mathcal{TTCP}^h$. Thus, id_c gets added to *used-id*$_c$, *crash-id*$_c$ is assigned `nil`, and *last-msg*$_c$ is assigned `null`.

In the *passive-open* step $last\text{-}msg_s$ is assigned `null`.

When the client performs the step with the $send\text{-}seg_{cs}(SYN,\ cc\_send,\ sn_c,\ msg_c)$ or $send\text{-}seg_{cs}(SYN,\ cc\_send,\ sn_c,\ msg_c,\ FIN)$ actions, if $msg_c \neq$ `null` then *cc_send* is added to the set *sent-tao-cc*. When the server receives either of these segments, it increments its sequence number, assigns the incremented value to isn_s, and sets $choose\text{-}isn_s$ to true. If the TAO test is successful, then the received *cc_send* value becomes the new id_s value of the server. Additionally, all the assignments that take place when a new id_s value is assigned at the server in specification $\mathcal{WD}$, happen in this step. In this situation the connection is also now established, so the pair (id_s, id_s) is added to *assoc*. We use this pair because $id_s = [cc\_send]$, and we know that $[cc\_send]$ is the id_c value of the client when the segment was sent. However, it might not be the id_c value of the client when the segment is received. Thus, the id_c and id_s value that form an association pair may not overlap. In TCP the initial sequence number pairs that form an association always overlap in time.

If the TAO test is not successful when the server receives one of these segments, it responds with the $send\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s)$ step. In this step $choose\text{-}isn_s$ is assigned to `false`. When the client receives this segment, if the acknowlegment number on the segment correctly acknowledges the sequence number of the client, then the pair $(id_c, [sn_s])$ is added to the set *estb-cc*. In response to this segment, the client performs the $send\text{-}seg_{cs}(cc\_send,\ sn_c,\ ack_c,\ msg_c)$ or $send\text{-}seg_{cs}(cc\_send,\ sn_c,\ ack_c,\ msg_c,\ FIN)$ step as the third leg of the three-way handshake. When the server receives either of these segments, if it completes the three-way handshake, the server assigns id_s to *cc_rcvd* and adds the pair (id_s, id_s) to *assoc*. The server also makes the other assignments associated with choosing a new id_s value.

In the $receive\text{-}msg_c(m)$ and $receive\text{-}msg_s(m)$ steps, $last\text{-}msg_c$ and $last\text{-}msg_s$ respectively are assigned the message m. In the crash and reset actions, the id's at the time of the crash or reset is added to the set of crash id's and and removed from the set of used id's.

send-msg$_c$*(open, m, close)*
Eff: (* Effect clause from $\mathcal{TTCP}_c$ *)
    if $mode_c$ = `closed` $\wedge$ *open* then {
        ...
        $id_c := cc\_send$
        $used\text{-}id_c := used\text{-}id_c \cup \{id_c\}$
        if $id_c \neq crash\text{-}id_c \wedge$
        $crash\text{-}id_c \neq$ `nil` then
            $used\text{-}id_c := used\text{-}id_c \cup \{crash\text{-}id_c\}$
        $crash\text{-}id_c :=$ `nil`
        $last\text{-}msg_c :=$ `null`
        ...

send-seg$_{cs}$*(SYN, cc_send, sn*$_c$*, msg*$_c$*)*
Pre: (* Precondition clause from $\mathcal{TTCP}_c$ *)
Eff: (* Effect clause from $\mathcal{TTCP}_c$ *)
    if $msg_c \neq$ `null` then
        $sent\text{-}tao\text{-}cc := sent\text{-}tao\text{-}cc \cup \{cc\_send\}$
    ...

send-seg$_{cs}$*(SYN, cc_send, sn*$_c$*, msg*$_c$*, FIN)*
Pre: (* Precondition clause from $\mathcal{TTCP}_c$ *)
Eff: (* Effect clause from $\mathcal{TTCP}_c$ *)
    if $msg_c \neq$ `null` then
        $sent\text{-}tao\text{-}cc := sent\text{-}tao\text{-}cc \cup \{cc\_send\}$
    ...

passive-open
Eff: if $mode_s$ = `closed` then {
    $last\text{-}msg_s :=$ `null`
    ...

receive-seg$_{cs}$*(SYN, cc_send, sn*$_c$*, msg*$_c$*)*
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
    if $mode_s$ = `listen` then {
        $sn_s := sn_s + 1$
        $isn_s := sn_s$
        $choose\text{-}isn_s :=$ `true`
        ...
        if $cache\_cc < cc\_send$ then {
            $id_s := cc\_send$
            $used\text{-}id_s := used\text{-}id_s \cup \{id_s\}$
            if $id_s \neq crash\text{-}id_s \wedge$
            $crash\text{-}id_s \neq$ `nil` then
                $used\text{-}id_s := used\text{-}id_s \cup \{crash\text{-}id_s\}$
            $crash\text{-}id_s :=$ `nil`
            $assoc := assoc \cup \{(id_s, id_s)\}$
            ...

receive-seg$_{cs}$*(SYN, cc_send, sn*$_c$*, msg*$_c$*, FIN)*
Eff: (* Effect clause from $\mathcal{TCP}_s$ *)
    if $mode_s$ = `listen` then {
        $sn_s := sn_s + 1$
        $isn_s := sn_s$
        $choose\text{-}isn_s :=$ `true`
        ...
        if $cache\_cc < cc\_send$ then {
            $id_s := cc\_send$
            $used\text{-}id_s := used\text{-}id_s \cup \{id_s\}$
            if $id_s \neq crash\text{-}id_s \wedge$
            $crash\text{-}id_s \neq$ `nil` then
                $used\text{-}id_s := used\text{-}id_s \cup \{crash\text{-}id_s\}$
            $crash\text{-}id_s :=$ `nil`
            $assoc := assoc \cup \{(id_s, id_s)\}$
            ...

Figure 10-6: *Changes made to $\mathcal{TTCP}_c$ and $\mathcal{TTCP}_s$ to get history T/TCP.*

receive-seg$_{sc}$*(SYN, cc_rcvd,* sn_s*,* ack_s*)* `Eff`: (* Effect clause from $\mathcal{TTCP}_c$ *) ... else { *send-ack*$_c$:= `true` if $mode_c \in \{$`syn-sent`, `syn-sent*`$\}$ then { $estb\text{-}cc := estb\text{-}cc \cup \{(id_c, sn_s)\}$ } ...	*send-seg*$_{sc}$*(SYN, cc_rcvd,* sn_s*,* ack_s*)* `Pre`: (* Precondition clause from $\mathcal{TTCP}_s$ *) `Eff`: (* Effect clause from $\mathcal{TTCP}_c$ *) *choose-isn*$_s$:= `false`
send-seg$_{cs}$*(cc_send,* sn_c*,* ack_c*,* msg_c*)* `Pre`: (* Precondition clause from $\mathcal{TTCP}_c$ *) `Eff`: (* Effect clause from $\mathcal{TTCP}_c$ *) ...	*receive-seg*$_{cs}$*(cc_send,* sn_c*,* ack_c*,* msg_c*)* `Eff`: (* Effect clause from $\mathcal{TTCP}_s$ *) ... else if $mode_s \neq$ `rec` $\wedge\ cc\_send\ = cc\_rcvd$ then { ... if $ack_c = sn_s + 1$ then { ... if $mode_s =$ `syn-rcvd` then { $id_s := cc\_send$ $used\text{-}id_s := used\text{-}id_s \cup \{id_s\}$ if $id_s \neq crash\text{-}id_s\ \wedge$ $crash\text{-}id_s \neq$ `nil` then $used\text{-}id_s := used\text{-}id_s \cup \{crash\text{-}id_s\}$ $crash\text{-}id_s :=$ `nil` if $\{(id_s, id_s)\} \notin assoc$ then { $assoc := assoc \cup \{(id_s, id_s)\}$ ...
send-seg$_{cs}$*(cc_send,* sn_c*,* ack_c*,* msg_c*, FIN)* `Pre`: (* Precondition clause from $\mathcal{TTCP}_c$ *) `Eff`: (* Effect clause from $\mathcal{TTCP}_c$ *) ...	*receive-seg*$_{cs}$*(cc_send,* sn_c*,* ack_c*,* msg_c*, FIN)* `Eff`: (* Effect clause from $\mathcal{TCP}_s$ *) ... else if $mode_s \neq$ `rec` $\wedge\ cc\_send\ = cc\_rcvd$ then { ... if $ack_c = sn_s + 1$ then { ... if $mode_s =$ `syn-rcvd` $\wedge$ $id_s := cc\_send$ $used\text{-}id_s := used\text{-}id_s \cup \{id_s\}$ if $id_s \neq crash\text{-}id_s\ \wedge$ $crash\text{-}id_s \neq$ `nil` then $used\text{-}id_s := used\text{-}id_s \cup \{crash\text{-}id_s\}$ $crash\text{-}id_s :=$ `nil` if $\{(id_s, id_s)\} \notin assoc$ then { $assoc := assoc \cup \{(id_s, id_s)\}$ ...

Figure 10-7: *Other changes made to $\mathcal{TTCP}$ for history T/TCP.*

$\mathit{receive\text{-}msg}_c(m)$ **Pre**: (* Precondition clause from $\mathcal{TTCP}_c$ *) **Eff**: (* Effect clause from $\mathcal{TTCP}_c$ *) $\mathit{last\text{-}msg}_c := m$ ... $\mathit{receive\text{-}seg}_{sc}(RST,\ ack_s,\ rst\text{-}seq_s)$ **Eff**: if $mode_c \neq \texttt{rec} \wedge rst\text{-}seq_s = ack_c \vee$ $(rst\text{-}seq_s = 0 \wedge ack_s = sn_c + 1)$ $mode_c := \texttt{reset}$ $crash\text{-}id_c := id_c$ $used\text{-}id_c := used\text{-}id_c \setminus id_c$ $crash_c$ **Eff**: (* Effect clause from $\mathcal{TTCP}_c$ *) if $mode_c \neq \texttt{closed}$ then ... $crash\text{-}id_c := id_c$ $used\text{-}id_c := used\text{-}id_c \setminus id_c$	$\mathit{receive\text{-}msg}_s(m)$ **Pre**: (* Precondition clause from $\mathcal{TTCP}_s$ *) **Eff**: (* Effect clause from $\mathcal{TTCP}_s$ *) $\mathit{last\text{-}msg}_s := m$ ... $\mathit{receive\text{-}seg}_{cs}(RST,\ ack_c,\ rst\text{-}seq_c)$ **Eff**: if $mode_s \neq \texttt{rec} \wedge rst\text{-}seq_c = ack_s$ then $mode_s := \texttt{reset}$ if $id_s \neq \texttt{nil}$ then $crash\text{-}id_s := id_s$ $used\text{-}id_s := used\text{-}id_s \setminus id_s$ $crash_s$ **Eff**: (* Effect clause from $\mathcal{TCP}_s$ *) if $mode_s \neq \texttt{closed}$ then ... if $id_s \neq \texttt{nil}$ then $crash\text{-}id_s := id_s$ $used\text{-}id_s := used\text{-}id_s \setminus id_s$

Figure 10-8: *The last set of changes made to $\mathcal{TTCP}$ for history T/TCP.*

10.2.2 Derived variables for $\mathcal{TTCP}^h$

We define several derived variables for $\mathcal{TTCP}^h$. They are used in the formal verification of the protocol. The first two derived variables we define for $\mathcal{TTCP}^h$ are $cur\text{-}msg_c$ and $cur\text{-}msg_s$. These are the "current message" being sent by the client and server respectively that have not yet been received. They are similar to the derived variables with the same names defined for $\mathcal{TCP}$. One main difference between these variables and the variables defined for $\mathcal{TCP}$, is that if the receiving host crashes or resets after the current message is received, but before it is acknowledged, the variable goes from being empty to the value it had before the message was received. The reason for this is that, after a crash or reset, the receiving host when it reopens, may accept this message again. This is the duplicate delivery that is allowed in our weaker specification. The variables are formally defined as follows.

$$
s.cur\text{-}msg_c \triangleq \begin{cases} (s.msg_c, \mathtt{ok}) & \text{if } s.mode_c \notin \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}\} \ \wedge s.msg_c \neq \mathtt{null} \ \wedge \\ & \quad ((s.cc\_send > s.cache\_cc) \wedge (s.mode_c \in \{\mathtt{syn\text{-}sent}, \mathtt{syn\text{-}sent*}\})) \ \vee \\ & \quad ((s.cc\_send = s.cc\_rcvd) \ \wedge \ ((s.sn_c = s.ack_s + 1) \ \vee \\ & \quad (\neg(s.rcvd\text{-}close_c \wedge s.send\text{-}buf_c \ = \epsilon) \ \wedge s.sn_c = s.ack_s))) \\ (s.msg_c, \mathtt{marked}) & \text{if } s.mode_c \notin \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}\} \ \wedge s.msg_c \neq \mathtt{null} \ \wedge \\ & \quad ((s.id_c, j) \in s.estb\text{-}cc \ \wedge (s.isn_s \neq j \ \vee \ s.mode_s \in \{\mathtt{rec}, \mathtt{reset}\})) \ \vee \\ & \quad ((s.id_c, l) \in s.assoc \ \wedge (s.id_s \neq l \vee s.mode_s \in \{\mathtt{rec}, \mathtt{reset}\})) \\ \epsilon & \text{otherwise} \end{cases}
$$

$$
s.cur\text{-}msg_s \triangleq \begin{cases} (s.msg_s, \mathtt{ok}) & \text{if } s.mode_s \notin \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}, \mathtt{listen}, \mathtt{syn\text{-}rcvd}\} \ \wedge s.msg_s \neq \mathtt{null} \\ & \quad \wedge \ ((s.cc\_rcvd = s.cc\_send) \ \wedge \ (s.mode_c \in \{\mathtt{syn\text{-}sent}, \mathtt{syn\text{-}sent*}\} \ \wedge \\ & \quad s.mode_s \in \{\mathtt{estb*}, \mathtt{fin\text{-}wait1*}, \mathtt{close\text{-}wait*}, \mathtt{last\text{-}ack*}\}) \ \vee \\ & \quad (\neg(s.rcvd\text{-}close_s \wedge s.send\text{-}buf_s \ = \epsilon) \ \wedge (s.sn_s = s.ack_c)) \\ & \quad (s.sn_s = s.ack_c + 1))) \ \vee \\ (s.msg_s, \mathtt{marked}) & \text{if } s.mode_s \notin \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}, \mathtt{listen}, \mathtt{syn\text{-}rcvd}\} \ \wedge s.msg_s \neq \mathtt{null} \\ & \quad \wedge \ ((k, s.id_s) \in s.assoc \ \wedge (s.mode_c \in \{\mathtt{rec}, \mathtt{reset}\} \ \vee \ s.id_c \neq k)) \\ \epsilon & \text{otherwise} \end{cases}
$$

In the formal definition for $cur\text{-}msg_c$, we have the condition that $mode_c \notin \{$`rec`, `reset`, `closed`$\} \wedge \ msg_c \neq$ `null`, because the client does not send messages if these conditions are true. When the client is not in one of these modes, there are three basic types of situations where the client is ready to send or is sending valid data that has not yet been received. The first situation occurs when the message is being sent on a SYN segment. In this situation $s.mode_c \in \{\mathtt{syn\text{-}sent}, \mathtt{syn\text{-}sent*}\}$ and the data is accepted at the server if $s.cc\_send > s.cache\_cc$. If the client is not sending a SYN segment, then in order for the data on the segment to be accepted, $s.cc\_send$ must be equal to $s.cc\_rcvd$. If the non-SYN segment is non-FIN segment; that is, $\neg(s.rcvd\text{-}close_c \wedge s.send\text{-}buf_c \ = \epsilon)$, then the data is accepted at the server if s.$sn_c = s.ack_s$. If the segment is a FIN segment, then the data is accepted if s.$sn_c = s.ack_s + 1$. These, are the "normal" conditions for the existence of the $cur\text{-}msg_c$ variable, and the messages are paired with the value `ok`, to match variables on the queues in $\mathcal{WD}$.

In the situation mentioned above in which the server crashes or resets after receiving the message that the client is sending, neither of the three conditions holds. However, in this situation we want the $cur\text{-}msg_c$ variable to have the value it had before the message was received at the server. Thus, we have the additional conditions of $((s.id_c, j) \in s.estb\text{-}cc \wedge (s.isn_s \neq j \vee s.mode_s \in \{\texttt{rec}, \texttt{reset}\})) \vee ((s.id_c, l) \in s.assoc \wedge (s.id_s \neq l \vee s.mode_s \in \{\texttt{rec}, \texttt{reset}\}))$ for the existence of this variable. The $(s.id_c, j) \in s.estb\text{-}cc$ and $(s.id_c, l) \in s.assoc$ parts of this condition are needed because one or both of these conditions must hold in order for a message from the client to have previously being delivered to the server. The $(s.isn_s \neq j \vee s.mode_s \in \{\texttt{rec}, \texttt{reset}\})$ and $(s.id_s \neq l \vee s.mode_s \in \{\texttt{rec}, \texttt{reset}\})$ parts of the condition reflects the fact the server may close and reopen after the crash or reset, but the current message may still be delivered. However, since the message may not get delivered if the client receives an acknowlegment, the message is paired with `marked` in this situation.

The definition for the server side version of the current message derived variable is essentially symmetric. However, the server has a valid message only if $s.mode_s \notin \{$`rec`, `reset`, `closed`, `listen`, `syn-rcvd`$\}$ and $s.cc\_rcvd = s.cc\_send$; and it sends a valid SYN segment with data that may get accepted if $s.mode_s \in \{$`estb*`, `fin-wait1*`, `close-wait*`, `last-ack*`$\}$ and $s.mode_c \in \{$`syn-sent`, `syn-sent*`$\}$.

The next two variables we define are $p\text{-}triple_c(k)$ and $p\text{-}triple_s$. These variables are "possible triples" and are similar to the "possible pairs" defined for $\mathcal{TCP}$. The term "triple" is used because the variables are sets of triples of the form (k, i, m) where k is a connection count, i is a sequence number, and m is a message. Like the possible pairs of $\mathcal{TCP}$, the possible triples of $\mathcal{TTCP}^h$ represents segments that contain messages that might get delivered after the sender crashes or receives a reset. Another way of looking at possible triples is that they represents the segments that contain the messages from the "current message" derived variable, after the sender crashed or reset. However, whereas in $\mathcal{TCP}$ there is at most one segment from the client (and duplicates of it) that may still be delivered in this situation, in $\mathcal{TTCP}^h$ there may be several of these segments. Thus, for $\mathcal{TTCP}^h$ the possible triple variable for segments from the client to the server is an array indexed by the connection count of the segment. For segments from the server to the client there is a most one possible triple, so this variable is not an array. The executions that can cause several different possible triples from the client to the server are executions where the client sends a segment with data that will pass the TAO test if it arrives at the server, crashes or resets after sending this segment, and then reopens and send a new segment with an incremented connection count that will also pass the TAO test if it arrives at the server. This process may be repeated several times, and each time it is repeated a new segment is added that can be delivered.

For any segment p on $\mathit{in\text{-}transit}_{cs}$ or $\mathit{in\text{-}transit}_{sc}$, recall that in Chapter 6 we define $sn(p)$ to be the sequence number of the segment, $ack(p)$ to be the acknowledgment number of the segment, and $msg(p)$ is the message of the segment. Segments sent by the client and server in $\mathcal{TTCP}^h$ also have connection counts. For segments from the client this is the *cc_send* value, and for segments from the server this is the *cc_rcvd* value. For a segment $p \in \mathit{in\text{-}transit}_{cs}$, $\mathit{cc\_send}(p)$ is the value of *cc_send* on that segment, and if $p \in \mathit{in\text{-}transit}_{sc}$, $\mathit{cc\_rcvd}(p)$ is the value of *cc_rcvd* on that segment.

Let s be any state in $\mathcal{TTCP}^h$. Then define *possible triples* as follows.

$$s.\mathit{p\text{-}triple}_c(k) \triangleq \begin{cases} \{(k,i,m) \mid \exists\, p \in s.\mathit{in\text{-}transit}_{cs} \text{ s.t. } \mathit{ccount}(p) = k \wedge sn(p) = i \;\wedge msg(p) = m\} \\ \quad \text{if } (s.\mathit{mode}_c \in \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}\} \;\vee s.\mathit{cc\_send} \;\neq k) \wedge \\ \quad (k > \mathit{cache\_cc} \;\wedge\; m \neq \mathtt{null} \;\wedge\; p \text{ is a SYN segment}) \vee \\ \quad (k = s.\mathit{cc\_rcvd} \;\wedge\; i = s.\mathit{ack}_s \;\wedge\; p \text{ is not a FIN or SYN segment}) \vee \\ \quad (k = s.\mathit{cc\_rcvd} \;\wedge\; i = s.\mathit{ack}_s + 1 \;\wedge\; p \text{ is a non-SYN FIN segment}) \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$s.\mathit{p\text{-}triple}_s \triangleq \begin{cases} \{(l,j,m) \mid \exists\, p \in s.\mathit{in\text{-}transit}_{sc} \text{ s.t. } \mathit{ccount}(p) = l \wedge sn(p) = j \wedge msg(p) = m\} \\ \quad \text{if } s.\mathit{mode}_s \in \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}, \mathtt{listen}, \mathtt{syn\text{-}rcvd}\} \;\wedge\; l = s.\mathit{cc\_send} \;\wedge \\ \quad (s.\mathit{mode}_c \in \{\mathtt{syn\text{-}sent}, \mathtt{syn\text{-}sent*}\} \;\wedge\; m \neq \mathtt{null} \;\wedge\; p \text{ is a SYN segment}) \vee \\ \quad (j = s.\mathit{ack}_c \;\wedge\; p \text{ is not a FIN or SYN segment}) \vee \\ \quad (j = s.\mathit{ack}_c + 1 \wedge\; p \text{ is a non-SYN FIN segment}) \\ \emptyset \quad \text{otherwise} \end{cases}$$

In the formal definition for $s.\mathit{p\text{-}triple}_c(k)$, the fact that the client must have crashed or reset after sending segment p is indicated by the condition $s.\mathit{mode}_c \in \{\mathtt{rec}, \mathtt{reset}, \mathtt{closed}\} \vee s.\mathit{cc\_send} \;\neq k$. The case where $s.\mathit{cc\_send} \;\neq k$ occurs if the client closes and re-opens after the crash or reset. There are three types of segments from which a possible triple variable can be derived. The three types here are basically segments that are sent in the three situations we have for current messages. The first type is SYN segments. For a SYN segment that has a message that is not `null`, that message may still be accepted if $k > s.\mathit{cache\_cc}$. The second type is segments that are not SYN or FIN segments. These segments may be accepted if $(k = s.cc\_rcvd \ \wedge \ i = s.ack_s)$. The third type is non-SYN that are FIN segments. Theses segments have acceptable data if $(k = s.cc\_rcvd \ \wedge \ i = s.ack_s + 1)$.

For $s.p\text{-}triple_s$, the conditions are slightly different. First, there is no situation where there are multiple different segments with messages from the server that the client may accept after the server crashes or resets. This is because the client always initiates the communication for a connection, and the TAO test only happens at the server. Second, the client only accepts the message on the segment if $l = s.cc\_send$. After the server crashes or resets, the message may still be delivered if $mode_s \in \{\texttt{rec}, \texttt{reset}, \texttt{closed}, \texttt{listen}, \texttt{syn-rcvd}\}$. If the server is in any other mode, it means it has started a new incarnation with the client, so the client will not accept messages from the previous incarnation. The basic three types of segments for which this possible triple variable is derived is the same.

The fifth derived variable is the *temporary message*. The formal definition follows.

$$s.temp\text{-}msg \triangleq \begin{cases} (s.temp\text{-}data, \texttt{ok}) & \text{if } s.temp\text{-}data \neq \texttt{null} \\ \epsilon & \text{otherwise} \end{cases}$$

This derived variable is similar to the current message, in that it is a single message paired with `ok` or it is the empty string. This variable is only defined for the server side, and it is the pairing the message in *temp-data* with `ok` until the message is added to the server's receive buffer. After that temporary message becomes the empty string.

10.3 Invariants of $\mathcal{TTCP}^h$

As we did for the other simulation proofs in this thesis, we need to prove a set of invariants on the reachable states of $\mathcal{TTCP}^h$ in order to limit the states we need to consider for the simulation proof. For $\mathcal{TTCP}$, the set of synchronized states for the server side grows to include the set of partially synchronized states. Thus, in the statement of the invariants the set *sync-states* = {`estb`, `fin-wait-1`, `fin-wait-2`, `close-wait`, `last-ack`, `timed-wait`, `estb*`, `fin-wait1*`, `close-wait*`, `closing*`, `last-ack*`}. We do not present the proofs for these invariants, but the proofs are similar to the proofs in B.

Invariants 10.7, 10.8, and 10.9 state some basic properties of sequence numbers, connection count numbers, and the sets *estb-cc*, and *assoc*.

Invariant 10.7

1. For all segments $p \in \mathit{in\text{-}transit}_{cs}$, $sn_c \geq sn(p)$.

2. For all segments $p \in \mathit{in\text{-}transit}_{sc}$, $sn_s \geq sn(p)$.

3. For all segments $p \in \mathit{in\text{-}transit}_{cs}$, $\mathit{cc\_send} \geq \mathit{cc\_send}(p)$. ■

Invariant 10.8

1. If $\mathit{mode}_c \neq \texttt{closed}$ then $\mathit{cc\_send} = \mathit{id}_c$.

2. If $\mathit{mode}_s \in \mathit{sync\text{-}states}$ then $\mathit{cc\_rcvd} = \mathit{id}_s$.

3. If there exists a segment $p \in \mathit{in\text{-}transit}_{cs}$ such that $\mathit{cc\_send}(p) = k$, then $k \in \mathit{used\text{-}id}_c \cup \{\mathit{crash\text{-}id}_c\}$. ■

Invariant 10.9

1. For all $i \in \mathtt{N} \cup \{\texttt{nil}\}$, $(i, \texttt{nil}) \notin \mathit{estb\text{-}cc}$.

2. For all $j \in \mathtt{N} \cup \{\texttt{nil}\}$, $(\texttt{nil}, j) \notin \mathit{estb\text{-}cc}$.

3. For all $i \in \mathtt{N} \cup \{\texttt{nil}\}$, $(i, \texttt{nil}) \notin \mathit{assoc}$.

4. For all $j \in \mathtt{N} \cup \{\texttt{nil}\}$, $(\texttt{nil}, j) \notin \mathit{assoc}$. ■

Invariant 10.10 states that the values of some server side variables are determined by the value of mode_s.

Invariant 10.10

1. If $\mathit{mode}_s \in \{\texttt{listen}, \texttt{syn-rcvd}\}$ then $\mathit{rcv\text{-}buf}_s = \epsilon$.

2. If $\mathit{mode}_s \in \{\texttt{listen}, \texttt{syn-rcvd}\}$ then $\mathit{msg}_s = \texttt{null}$.

3. If $\mathit{mode}_s \in \{\texttt{listen}, \texttt{syn-rcvd}\}$ then $\mathit{last\text{-}msg}_s = \texttt{null}$.

4. If $\mathit{mode}_s \in \mathit{sync\text{-}states}$ then $\mathit{temp\text{-}data} = \texttt{null}$. ■

Invariant 10.11 states conditions under which the id's of the client and/or the server, and the initial sequence number of the server, are not part of an *assoc* or an *estb-cc* pair. Invariant 10.12 does the opposite, it states conditions under which we know the id's of the client and/or the server, and the initial sequence number of the server, are part of an *assoc* or an *estb-cc* pair.

Invariant 10.11

1. If $mode_s = \texttt{listen}$ and there exists a segment $p \in in\text{-}transit_{cs}$ such that $cc\_send(p) > cache\_cc$ then $(cc\_send(p), cc\_send(p)) \notin assoc$.

2. If $mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\}$ then for all j, $(id_c, j) \notin estb\text{-}cc$.

3. If $mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\}$ and there exists a segment p of the form *(SYN, cc_rcvd, sn_s, ack_s)* or *(SYN, cc_rcvd, sn_s, ack_s, FIN)* in $in\text{-}transit_{sc}$ such that $cc\_rcvd(p) = cc\_send$ and $ack(p) = sn_s + 1$ then for all j, $(id_c, j) \notin assoc$.

4. If $mode_s = \texttt{syn-rcvd} \wedge choose\text{-}isn_s$ then for all k, $(k, isn_s) \notin estb\text{-}cc$. ∎

Invariant 10.12

1. If $mode_s = \texttt{syn-rcvd}$ and there exists a segment $p \in in\text{-}transit_{cs}$ such that $cc\_send(p) = cc\_rcvd \wedge ack(p) = sn_s + 1$, then $(cc\_send(p), isn_s) \in estb\text{-}cc$.

2. If $mode_c \in sync\text{-}states$ or there exists a segment p of type *(SYN, cc_rcvd, sn_s, ack_s, msg_s,)* or *(SYN, cc_rcvd, sn_s, ack_s, msg_s, FIN)* with $cc\_rcvd(p) = ccsend$ then $(id_c, id_c) \in assoc$.

3. If $mode_s \in sync\text{-}states$ then $(id_s, id_s) \in assoc$.

4. If $(k, id_s) \in assoc$ and there exists j such that $(k, j) \in estb\text{-}cc$ then $j = isn_s$. ∎

Invariants 10.13 and 10.14 are about the open phase of the protocol. They basically say that unless there is a crash or reset, the client and server are not out of synch.

Invariant 10.13

If $mode_s \in sync\text{-}states \wedge cc\_send = cc\_rcvd$ and there exists a non-SYN segment $p \in in\text{-}transit_{cs}$ with $cc\_send(p) = cc\_rcvd \wedge ack(p) = sn_s + 1$ then $mode_c \notin \{\texttt{syn-sent}, \texttt{syn-sent*}\}$. ∎

Invariant 10.14

If $mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\}$ and there exists a SYN segment $p \in \mathit{in\text{-}transit}_{sc}$ such that $\mathit{cc\_rcvd}(p) = \mathit{cc\_send} \wedge ack(p) = sn_c{+}1 \wedge sn(p) = isn_s$ then $mode_s \in \{\texttt{syn-rcvd}, \texttt{rec}, \texttt{reset}\}$.

■

Invariant 10.15 states that if a host has started the close phase (indicated by its mode), it must have received the signal to close from the user ($\mathit{rcvd\text{-}close}_c$ or $\mathit{rcvd\text{-}close}_s$ is `true`), and it must have sent all the data it received from the user (the send buffers are empty).

Invariant 10.15

1. If $mode_c \in \{\texttt{syn-sent*}, \texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$ then $\mathit{send\text{-}buf}_c = \epsilon \wedge \mathit{rcvd\text{-}close}_c = \texttt{true}$.

2. If $mode_s \in \{\texttt{fin-wait-1}, \texttt{fin-wait1*}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{closing*}, \texttt{timed-wait}, \texttt{last-ack}, \texttt{last-ack*}\}$ then $\mathit{send\text{-}buf}_s = \epsilon \wedge \mathit{rcvd\text{-}close}_s = \texttt{true}$. ■

Invariants 10.16 and 10.17 are about the relationship between msg_c and *temp-data*. They state that in certain states $msg_c = \mathit{temp\text{-}data}$. They are important because, in situations where *temp-data* gets lost because of a crash or reset. They show that the message that *temp-data* held may still be delivered.

Invariant 10.16

If $\mathit{temp\text{-}data} \neq \texttt{null} \wedge msg_c \neq \texttt{null} \wedge ((id_c, isn_s) \in \mathit{estb\text{-}cc} \vee (id_c, id_s) \in \mathit{assoc}) \wedge sn_c < ack_s$ then $msg_c = \mathit{temp\text{-}data}$. ■

Invariant 10.17

If $mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\} \wedge \mathit{temp\text{-}data} \neq \texttt{null}$ and there exists a SYN segment $p \in \mathit{in\text{-}transit}_{sc}$ such that $\mathit{cc\_rcvd}(p) = \mathit{cc\_send} \wedge ack(p) = sn_c + 1 \wedge sn(p) = isn_s$ then $msg_c = \mathit{temp\text{-}data}$. ■

Invariants 10.18, 10.19, and 10.20 state properties that are important for the situations when a host crashes or resets. Informally, Invariant 10.18 says that if all the messages received by a host has been passed to the user, then the last message passed to the user is the same as the last message sent by the sending host. Invariant 10.19, says that if all the messages have not been passed to the user, then the message at the back of the receive buffer is the same as the last message sent by the sending host. Invariant 10.20 says that if there has not been a message passed to the user as yet, but the sender sent a message that is received, then that message must still be on the receive buffer.

Invariant 10.18

1. If $\mathit{rcv\text{-}buf}_c = \epsilon \ \wedge \ \mathit{last\text{-}msg}_s \neq \texttt{null} \ \wedge \ \mathit{msg}_s \neq \texttt{null} \ \wedge \ (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_s < \mathit{ack}_c$ then $\mathit{msg}_s = \mathit{last\text{-}msg}_c$.

2. If $\mathit{rcv\text{-}buf}_s = \epsilon \ \wedge \ \mathit{last\text{-}msg}_c \neq \texttt{null} \ \wedge \ \mathit{msg}_c \neq \texttt{null} \ \wedge \ (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_c < \mathit{ack}_s$ then $\mathit{msg}_c = \mathit{last\text{-}msg}_s$. ■

Invariant 10.19

1. If $\mathit{rcv\text{-}buf}_c \neq \epsilon \ \wedge \ \mathit{msg}_s \neq \texttt{null} \ \wedge \ (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_s < \mathit{ack}_c$ then $\mathit{msg}_s = \mathit{last}(\mathit{rbuf}_c)$.

2. If $\mathit{rcv\text{-}buf}_s \neq \epsilon \ \wedge \ \mathit{msg}_c \neq \texttt{null} \ \wedge \ (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_c < \mathit{ack}_s$ then $\mathit{msg}_s = \mathit{last}(\mathit{rbuf}_s)$. ■

Invariant 10.20

1. If $\mathit{last\text{-}msg}_c = \texttt{null} \wedge \mathit{msg}_s \neq \texttt{null} \wedge (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_s < \mathit{ack}_c$ then $\mathit{rcv\text{-}buf}_c \neq \epsilon$.

2. If $\mathit{last\text{-}msg}_s = \texttt{null} \wedge \mathit{msg}_c \neq \texttt{null} \wedge (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_c < \mathit{ack}_s$ then $\mathit{rcv\text{-}buf}_s \neq \epsilon$. ■

Invariants 10.21, 10.22, and 10.23 deal with properties of messages at the hosts and on segments. Invariant 10.21 states that if the message at a host is not `null`, and there is a segment with the same sequence number as the host, then the segment must have the same message as the host. Invariant 10.22 is another key invariant. It states that if two segments on the same channel have the same sequence number and the messages on the segments are not `null`, then they must have the same message. Invariant 10.23 states that segments that cause the the value of the message variable on the segment to be added to the receive buffer, contains valid messages. That is, they contain messages that are not `null`.

Invariant 10.21

1. If $msg_c \neq \texttt{null}$ and there exists $p \in \textit{in-transit}_{cs}$ such that $sn(p) = sn_c$ then $msg(p) = msg_c$.

2. If $msg_s \neq \texttt{null}$ and there exists $p \in \textit{in-transit}_{sc}$ such that $sn(p) = sn_s$ then $msg(p) = msg_s$. ∎

Invariant 10.22

1. If there exists segments p and q on $\textit{in-transit}_{cs}$ such that $sn(p) = sn(q) \ \wedge msg(p) \neq \texttt{null} \ \wedge \ msg(q) \neq \texttt{null}$ then $msg(p) = msg(q)$.

2. If there exists segments p and q on $\textit{in-transit}_{sc}$ such that $sn(p) = sn(q) \ \wedge msg(p) \neq \texttt{null} \ \wedge \ msg(q) \neq \texttt{null}$ then $msg(p) = msg(q)$. ∎

Invariant 10.23

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup \textit{sync-states}$ and there exists $p \in \textit{in-transit}_{cs}$ such that $sn(p) = ack_s$ then $msg(p) \neq \texttt{null}$.

2. If $mode_c \in \textit{sync-states}$ and there exists $p \in \textit{in-transit}_{sc}$ such that $sn(p) = ack_c$ then $msg(p) \neq \texttt{null}$. ∎

Invariant 10.24 states that whenever the client has an acknowledgment number, it is greater than or equal to the acknowledgment number of any segment on the out going channel of the client, and Invariant 10.25 states that under certain conditions the acknowledgment number at the server is always bigger than the acknowledgment number of any segment on the out going channel of the server.

Invariant 10.24

If $ack_c \in \mathtt{N}$ then for all $p \in \textit{in-transit}_{cs}$, $ack_c \geq ack(p)$. ∎

Invariant 10.25

If $mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\} \ \wedge \ (id_c, id_s) \in assoc \ \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ then for all segments $p \in \textit{in-transit}_{sc}$, $ack_s \geq ack(p)$. ∎

Invariant 10.26 expresses a key correctness property. It states that sequence numbers do not get changed until the data sent with that sequence number is acknowledged.

Invariant 10.26

1. If there exists a SYN segment $p \in \mathit{in\text{-}transit}_{cs}$ such that $\mathit{cc\_send}(p) = \mathit{cc\_send}$ and $\mathit{cc\_send}(p) > \mathit{cache\_cc}$ then $sn_c = sn(p)$.

2. If $mode_s \in \{\texttt{syn-rcvd}\} \cup \mathit{sync\text{-}states} \wedge mode_c \in \{\texttt{rec}, \texttt{reset}\} \cup \mathit{sync\text{-}states}$ and there exists $p \in \mathit{in\text{-}transit}_{cs}$ such that $(\mathit{cc\_send}(p), isn_s) \in \mathit{estb\text{-}cc} \wedge \mathit{cc\_send}(p) = \mathit{cc\_rcvd} \wedge sn(p) \geq ack_s$, then $sn_c = sn(p)$.

3. If $mode_c \in \mathit{sync\text{-}states} \wedge (isn_c, isn_s) \in \mathit{assoc}$ and there exists $p \in \mathit{in\text{-}transit}_{sc}$ such that $sn(p) \geq ack_c$, then $sn_s = sn(p)$. ■

Invariant 10.27 states that if a host is in a mode that indicates it has received a FIN segment, and its id is paired with the other host's id, then that other host must be in a mode that indicates that it sent the FIN segment. That is, if a host accepts a FIN segment, it must be a legitimate FIN segment for the current incarnation of the connection.

Invariant 10.27

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, id_s) \in \mathit{assoc}$ then $mode_s \in \{$`fin-wait-1`, `fin-wait1*`, `fin-wait-2`, `closing`, `closing*`, `timed-wait`, `last-ack`, `last-ack*`$\}$.

2. If $mode_s \in \{$`close-wait`, `close-wait*`, `closing`, `closing*`, `last-ack`, `last-ack*`, `timed-wait`$\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, isn_s) \in \mathit{estb\text{-}cc} \vee (id_c, id_s) \in \mathit{assoc}$ then $mode_c \in \{$`syn-sent*`, `fin-wait-1`, `fin-wait-2`, `closing`, `timed-wait`, `last-ack`$\}$.
   ■

Invariants 10.28 and 10.29 are similar. Invariant 10.28 states that when a host is in a mode that indicates that it received a FIN segment, then if the other host has not closed since sending the FIN segment, its sequence number is less than the acknowledgment number of the host that received the FIN segment. Invariant 10.29 states that in the same situation as Invariant 10.28, but where the sending host may have closed, the sequence number on any sent segment is less than the acknowledgment number of the host that received the FIN segment.

Invariant 10.28

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ $\wedge\ (id_c, id_s) \in assoc$ then $sn_s < ack_c$.

2. If $mode_s \in \{\texttt{close-wait}$, $\texttt{close-wait*}$, $\texttt{closing}$, $\texttt{closing*}$, $\texttt{last-ack}$, $\texttt{last-ack*}$, $\texttt{timed-wait}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, isn_s) \in estb\text{-}cc \vee (id_c, id_s) \in assoc$ then $sn_c < ack_s$. ■

Invariant 10.29

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists l such that $(id_c, l) \in assoc$ then for all non-SYN segments $p \in in\text{-}transit_{sc}$, $sn(p) < ack_c$.

2. If $mode_s \in \{\texttt{close-wait}$, $\texttt{close-wait*}$, $\texttt{closing}$, $\texttt{closing*}$, $\texttt{last-ack}$, $\texttt{last-ack*}$, $\texttt{timed-wait}\}$ and there exists k, such that $(k, isn_s) \in estb\text{-}cc \vee (k, id_s) \in assoc$ then for all non-SYN segments $p \in in\text{-}transit_{cs}$, $sn(p) < ack_c$. ■

Invariant 10.30 expresses a property that is important for the $p\text{-}triple_c(k)$ and $p\text{-}triple_s$ derived variables. The invariant states that when a host receives a segment that may have acceptable data ($sn(p) \geq ack_c$ or $sn(p) \geq ack_s$), then all other segments q on the channel have $sn(q) \leq sn(p)$. This means that if the message was a part of a possible triple, the set becomes empty after this message is received because when the segment is received the acknowledgment number of the receiving host is set to $sn(p) + 1$.

Invariant 10.30

1. If $mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\}$ and there exists j such that $(id_c, j) \in assoc$ and there exists a SYN segment $p \in in\text{-}transit_{sc}$ such that $cc\_rcvd(p) = cc\_send$ then for all segments $q \in in\text{-}transit_{sc}$ such that $cc\_rcvd(q) = cc\_send$, $sn(q) \leq sn(p)$.

2. If $mode_c \in sync\text{-}states$ and there exists j such that $(id_c, j) \in assoc$ and there exists a non-SYN segment $p \in in\text{-}transit_{sc}$ such that $cc\_rcvd(p) = cc\_send \ \wedge \ sn(p) \geq ack_c$, then for all non-SYN segments $q \in in\text{-}transit_{sc}$ $sn(q) \leq sn(p)$.

3. If $mode_s \in \{\texttt{syn-rcvd}\} \cup \textit{sync-states}$ and there exists i, such that $i = isn_c^s \wedge (i, isn_s) \in \textit{estb-pairs}$ and there exists a non-SYN segment $p \in \textit{in-transit}_{cs}$ such that $sn(p) \geq ack_s$, then for all non-SYN segments $q \in \textit{in-transit}_{cs}$ $sn(q) \leq sn(p)$. ■

Invariants 10.31 and 10.32 state that when a host closes from mode `last-ack` or `last-ack*` its receive buffer is empty. Invariant 10.31 is for the situation where the other host has not closed, and Invariant 10.32 is for the situation where the other hosts might have closed after the connection is formed.

Invariant 10.31

1. If $mode_c = \texttt{last-ack} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, id_s) \in assoc$ then $\textit{rcv-buf}_c = \epsilon$.

2. If $mode_s \in \{\texttt{last-ack}, \texttt{last-ack*}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, isn_s) \in \textit{estb-cc} \vee (id_c, id_s) \in assoc$ then $\textit{rcv-buf}_s = \epsilon$. ■

Invariant 10.32

1. If $mode_c = \texttt{last-ack}$ and there exists l such that $(id_c, l) \in assoc$ then $\textit{rcv-buf}_c = \epsilon$.

2. If $mode_s \in \{\texttt{last-ack}, \texttt{last-ack*}\}$ and there exists k, such that $(k, isn_s) \in \textit{estb-cc} \vee (k, id_s) \in assoc$ then $\textit{rcv-buf}_s = \epsilon$. ■

The final invariant, Invariant 10.33, states that if a host is in mode that indicates that it received a FIN segment, then the other host must either have the flag set that indicates it received a close signal from its user, or if the flag is not set to true, it must be because the host closed after sending the FIN segment.

Invariant 10.33

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists l such that $(id_c, l) \in assoc$ then $\textit{rcvd-close}_s = \texttt{true} \vee id_s \neq l$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{close-wait*}, \texttt{closing}, \texttt{closing*}, \texttt{last-ack}, \texttt{last-ack*}, \texttt{timed-wait}\}$ and there exists k such that $(k, isn_s) \in \textit{estb-cc} \vee (k, id_c) \in assoc$ then $\textit{rcvd-close}_c = \texttt{true} \vee id_c \neq k$. ■

The conjunction of all the above invariants is itself an invariant, and we call this invariant I_{TT}.

10.4 The simulation proof

In this section we define a mapping from states of $\mathcal{TTCP}^h$ to states of $\mathcal{WD}^p$, and then prove that it is a timed refinement mapping with respect to invariant I_{WD} and I_{TT}.

10.4.1 The refinement mapping

We define a function R_{TTWD} *states*($\mathcal{TTCP}^h$) to *states*($\mathcal{WD}^p$).

Definition 10.2 (Refinement Mapping From $\mathcal{TTCP}^h$ to $\mathcal{WD}^p$)
For our mapping the *CID* and *SID* are instantiated by the set of non-negative integers. If $s \in$ *states*($\mathcal{TTCP}^h$) then define R_{TTWD} to be the state $u \in$ *states*($\mathcal{WD}^p$) such that:

1. $u.now = s.now$
2. $u.id_c = s.id_c$
   $u.id_s = s.id_s$
3. $u.choose\text{-}sid = (s.mode_s \in \{\texttt{listen}, \texttt{syn-rcvd}\})$
4. $u.rec_c = (s.mode_c = \texttt{rec})$
   $u.rec_s = (s.mode_s = \texttt{rec})$
5. $u.abrt_c = (s.mode_c = \texttt{reset})$
   $u.abrt_s = (s.mode_s = \texttt{reset})$
6. $u.used\text{-}id_c = s.used\text{-}id_c$
   $u.used\text{-}id_s = s.used\text{-}id_s$
7. $u.crash\text{-}id_c = s.crash\text{-}id_c$
   $u.crash\text{-}id_s = s.crash\text{-}id_s$
8. $u.assoc = s.assoc$
9. $u.last\text{-}msg_c = s.last\text{-}msg_c$
   $u.last\text{-}msg_s = s.last\text{-}msg_s$
10. $$\begin{array}{lcll} u.mode_c & = & \texttt{active} & \text{if } s.rcvd\text{-}close_c = \texttt{false} \\ & = & \texttt{inactive} & \text{if } s.rcvd\text{-}close_c = \texttt{true} \vee mode_c = \texttt{closed} \\ u.mode_s & = & \texttt{active} & \text{if } s.rcvd\text{-}close_s = \texttt{false} \\ & = & \texttt{inactive} & \text{if } s.rcvd\text{-}close_s = \texttt{true} \vee mode_s = \texttt{closed} \end{array}$$
11. $$\begin{array}{lcll} u.q\text{-}stat_{cs}(k) & = & \texttt{live} & \text{if } (s.id_c = k \wedge \ \forall\ j(k,j) \notin s.estb\text{-}cc \ \wedge (k,k) \notin s.assoc) \vee ((k,k) \in \\ & & & s.assoc \wedge (s.id_s = k \vee s.id_c = k)) \vee ((k, s.isn_s) \in s.estb\text{-}cc \wedge s.cc\_rcvd = \\ & & & k) \vee (k \in s.sent\text{-}tao\text{-}cc \wedge (k,k) \notin s.assoc) \\ & = & \texttt{dead} & \text{otherwise} \\ u.q\text{-}stat_{sc}(l) & = & \texttt{live} & \text{if } (s.id_s = l) \vee ((s.id_c, l) \in s.assoc) \\ & = & \texttt{dead} & \text{otherwise} \end{array}$$

$$
\begin{array}{llll}
12. & u.queue_{cs}(k) = \epsilon & & \text{if } ((s.id_c \neq k) \wedge \\
& & & (\forall\ j\ (k,j) \notin s.estb\text{-}cc\ \wedge\ (k,k) \notin s.assoc) \vee \\
& & & ((k,k) \in s.assoc\ \wedge s.id_s \neq k) \vee \\
& & & ((k,j) \in s.estb\text{-}cc\ \wedge\ s.isn_s \neq j)) \vee \\
& & & (s.mode_s \in \{\mathtt{rec},\mathtt{reset}\} \wedge \\
& & & (s.mode_c \in \{\mathtt{rec},\mathtt{reset}\}\ \vee s.id_c \neq k)) \qquad \text{(A)}
\end{array}
$$

$$
\begin{array}{lll}
= & \text{concatenation of:} & \text{if } (s.id_c = k) \wedge \\
& \bullet\ s.cur\text{-}msg_c & ((\forall\ j\ (k,j) \notin s.estb\text{-}cc\ \wedge (k,k) \notin s.assoc) \vee \\
& \bullet(s.send\text{-}buf_c \times \mathtt{ok}) & (s.mode_c \notin \{\mathtt{rec},\mathtt{reset}\}) \wedge \\
& & ((k,j) \in s.estb\text{-}cc\ \wedge (s.isn_s \neq j\ \vee\ s.mode_s \in \{\mathtt{rec},\mathtt{reset}\})) \\
& & \vee\ ((k,k) \in s.assoc\ \wedge (s.id_s \neq k \vee s.mode_s \in \{\mathtt{rec},\mathtt{reset}\})) \\
& & \text{(B)}
\end{array}
$$

$$
\begin{array}{lll}
= & \text{concatenation of:} & \text{if } (s.id_c = k) \wedge (s.mode_c \notin \{\mathtt{rec},\mathtt{reset}\}) \wedge \\
& \bullet\ s.temp\text{-}msg & ((k, s.isn_s) \in s.estb\text{-}cc\ \vee (k, s.id_s) \in s.assoc)) \\
& \bullet(s.rcv\text{-}buf_s \times \mathtt{ok}) & \wedge\ (s.mode_s \notin \{\mathtt{rec},\mathtt{reset}\}) \qquad \text{(C)} \\
& \bullet s.cur\text{-}msg_c & \\
& \bullet(s.send\text{-}buf_c \times \mathtt{ok}) &
\end{array}
$$

$$
\begin{array}{lll}
= & \text{concatenation of:} & \text{if } (s.id_c \neq k \vee s.mode_c \in \{\mathtt{rec},\mathtt{reset}\}) \wedge \\
& \bullet\ s.temp\text{-}msg & ((k, s.isn_s) \in s.estb\text{-}cc\ \vee\ (k, s.id_s) \in s.assoc) \\
& \bullet(s.rcv\text{-}buf_s \times \mathtt{ok}) & \wedge\ (s.mode_s \notin \{\mathtt{rec},\mathtt{reset}\}) \qquad \text{(D)} \\
& \bullet(data(s.p\text{-}triple_c(k)) & \\
& \times\ \mathtt{marked}) &
\end{array}
$$

$$
\begin{array}{lll}
= & (data(s.p\text{-}triple_c(k)) & \text{if } (s.id_c \neq k \vee s.mode_c \in \{\mathtt{rec},\mathtt{reset}\}) \wedge \\
& \times\ \mathtt{marked}) & ((k, s.isn_s) \notin s.estb\text{-}cc\ \wedge\ (k, s.id_s) \notin s.assoc) \wedge \\
& & (k \in s.sent\text{-}tao\text{-}cc\ \wedge k > s.cache\_cc) \qquad \text{(E)}
\end{array}
$$

$$
\begin{array}{llll}
13. & u.queue_{sc}(l) = \epsilon & & \text{if } (s.id_s \neq l \wedge\ (s.id_c, l) \notin s.assoc) \vee \\
& & & (s.mode_c \in \{\mathtt{rec},\mathtt{reset}\} \wedge \\
& & & (s.mode_s \in \{\mathtt{rec},\mathtt{reset}\}\ \vee\ s.id_s \neq l)) \qquad \text{(A)}
\end{array}
$$

$$
\begin{array}{lll}
= & \text{concatenation of:} & \text{if } (s.id_s = l) \wedge\ ((l,l) \notin s.assoc) \vee \\
& \bullet s.cur\text{-}msg_s & (s.mode_s \notin \{\mathtt{rec},\mathtt{reset}\}\ \wedge (l,l) \in s.assoc \wedge \\
& \bullet(s.send\text{-}buf_s \times \mathtt{ok}) & (s.mode_c \in \{\mathtt{rec},\mathtt{reset}\}\ \vee\ s.id_c \neq l))) \qquad \text{(B)}
\end{array}
$$

$$
\begin{array}{lll}
= & \text{concatenation of:} & \text{if } (s.id_s\ =\ l) \wedge\ \ ((s.id_c, l)\ \in\ s.assoc) \wedge (s.mode_s\ \notin \\
& \bullet(s.rcv\text{-}buf_c \times \mathtt{ok}) & \{\mathtt{rec},\mathtt{reset}\}) \wedge (s.mode_c \notin \{\mathtt{rec},\mathtt{reset}\}) \qquad \text{(C)} \\
& \bullet s.cur\text{-}msg_s & \\
& \bullet(s.send\text{-}buf_s \times \mathtt{ok}) &
\end{array}
$$

$$
\begin{array}{lll}
= & \text{concatenation of:} & \text{if } (s.id_s \neq l \vee\ s.mode_s \in \{\mathtt{rec},\mathtt{reset}\}) \wedge\ (s.id_c, l) \in \\
& \bullet(s.rcv\text{-}buf_c \times \mathtt{ok}) & s.assoc\ \wedge s.mode_c \notin \{\mathtt{rec},\mathtt{reset}\} \qquad \text{(D)} \\
& \bullet(data(s.p\text{-}triple_s) & \\
& \times\ \mathtt{marked}) &
\end{array}
$$

We now present some intuition behind the mapping R_{TTWD}. The mapping is similar to the refinement mapping R_{TD} presented in Chapter 7. Most of the equations in the mapping are straightforward. The interesting cases are for $u.q\text{-}stat_{cs}(k)$, $u.q\text{-}stat_{sc}(l)$, $u.queue_{cs}(k)$, and $u.queue_{sc}(l)$.

There are four sets of states of $\mathcal{TTCP}^h$ for which we want $u.q\text{-}stat_{cs}(k)$ to be `live`, in the corresponding set of states of $\mathcal{WD}^p$. These sets are not disjoint. The first set of states occurs when the client opens and assigns id_c the value k. This corresponds to the situation where the client first opens in $\mathcal{WD}^p$ and chooses k from CID and makes the queue indexed by k `live`. This is before k is paired with any isn_s value and added to *estb-cc* or paired with itself and added to *assoc*. Once k has been paired with itself and added to *assoc*; that is, $(k, k) \in assoc$, then if the client still has $id_c = k$, it may still send data for incarnation k, or if the server has $id_s = k$, it may still receive data for this incarnation. Therefore, in this situation the abstract queue is still `live` if $id_c = k \vee id_s = k$. The third set of states occur when $(k, s.isn_s) \in s.estb\text{-}cc \ \wedge \ k = s.cc\_rcvd$, this condition may be true for the second set of states, but there are cases where this condition is true and the second is not and vice versa. This set of states represents the situation where the second phase of the three-way handshake has been successful, and the server may accept data from the client, even if $s.id_c \neq k$. The fourth set of states are states where the client sends a SYN segment for TAO ($k \in sent\text{-}tao\text{-}cc$), but the segment has not yet been received ($(k, k) \notin assoc$). The data on the abstract queues in this situation may or may not be deliverable.

The conditions for $u.q\text{-}stat_{sc}(l)$ to be `live` are much simpler. They are the symmetric situation to the first two set of cases for $u.q\text{-}stat_{cs}(k)$. That is, when the server first assigns id_s the value l, or if l is paired with the current id of the client. We have these simpler conditions because the server can only send messages for incarnation l if it has $id_c = l$, and the client only accepts data for this incarnation if $(s.id_c, l) \in s.assoc$.

For $u.queue_{cs}(k)$, there are five cases for the mapping of variables of $\mathcal{TTCP}^h$ to this variable. The first case corresponds to the states that map the status of the abstract queue to `dead`, or if the server has crash or reset, and the client has also crashed or reset or is closed after the crash or reset, or has reopened after the crash or reset ($id_c \neq k$). For this case we want the abstract queue to be empty.

The second case, (B), corresponds to two different sets of states. The first set of states occurs when the client has just open and its id_c value is not part of an *assoc* or *estb-cc* pair. In this situation, the abstract queue corresponds to just send buffer of the client, and the current message the client might be sending. The second set of states where the abstract queue again corresponds to the concatenation of $cur\text{-}msg_c$ and $(send\text{-}buf_c \times \texttt{ok})$ occurs when the server crashes or resets after id_s was paired with id_c, or its isn_s value was paired with id_c. In this situation, any part of the abstract queue that is represented by variables at the server are no longer deliverable, so these variables are lost. Thus, the abstract queue is only represent by the variables at the client. This is the situation where $cur\text{-}msg_c$ may go from being empty to having a message after the server crashes or resets that we discussed in the presentation on the $cur\text{-}msg_c$ derived variable in Section 10.2.2.

Case (C) is the normal message delivery situation. That is, the client has not crashed, reset or closed since it opened and assigned id_c value k for the incarnation, so $(s.id_c = k) \wedge\ s.mode_c \notin \{\mathtt{rec}, \mathtt{reset}\}$. Also the client has formed an association pair with the id of the server, $(k, s.id_s) \in s.assoc$, or is about form such a pair, $(k, s.isn_s) \in s.estb\text{-}cc$, and the server has not crashed or reset. For this situation, data that corresponds to parts of the abstract queue may be in the *temp-data*, the receive buffer at the server, msg_c, and the send buffer of the client. If the server crashes or resets in this situation, the mapping reverts to case (B). If the client crashes or resets, we get case (D).

For case (D), since the client has crashed or reset after being in case (C), the parts of the abstract queue that corresponded to variables on the client side are lost. However, the msg_c variable in $cur\text{-}msg_c$ may be on a segment on the channel that might still get delivered. This message is *data(*p-triple$_c$*(k))*. However, because there is a possibility it might not be delivered, it is paired with `marked`.

The fifth and final case, (E), is the set of states where a crash or reset occurs while $\mathcal{TTCP}^h$ is in the first set of states of case (B). This is also the situation where we get multiple possible triple variables for different incarnations that may be delivered to the server. These queues contain at most one element, and it is marked because it may not get delivered.

The four cases for the mapping to $u.queue_{sc}(l)$ are basically the symmetric counterparts to the first four cases for the the mapping to $u.queue_{cs}(k)$. The fact that the conditions are simpler reflect the fact the conditions for $u.queue_{sc}(l)$ being `live` are simpler than the conditions for $u.queue_{cs}(k)$ being `live`. They also reflect the fact and that there is no temporary data on the client side, and that there is only a one possible triple variable in any given state.

10.4.2 Simulation of steps

In this section we prove that the mapping R_{TTWD} defined in the previous section is indeed a timed refinement mapping from $\mathcal{TTCP}^h$ to $\mathcal{WD}^p$ with respect to I_{WD} and I_{TT}. This claim is stated as the following lemma.

Lemma 10.3
$\mathcal{TTCP}^h \leq_R^t \mathcal{WD}^p$ *via* R_{TTWD}.

Proof: We prove this lemma by showing that the two cases of Definition 3.11 are satisfied.

Base Case

In the start state s_0 of $\mathcal{TTCP}^h$ we have $s_0.mode_c = s_0.mode_s =$ `closed`, $s_0.now = 0$, and $s_0.assoc = \emptyset$. It is clear that $R_{\text{TTWD}}(s_0)$ is the unique start state u_0 of $\mathcal{WD}^p$.

Inductive Case

Assume $(s, a, s') \in$ *Steps*$(\mathcal{TTCP}^h)$. Below we consider cases based on a and for each case we define a finite execution fragment α of S such that *fstate*$(\alpha) = R_{\text{TTWD}}(s)$, *lstate*$(\alpha) = R_{\text{TTWD}}(s')$, and *t-trace*$(\alpha)$ = *t-trace*(s, a, s'). For the steps of the proof below we do not include the time of occurrence and last time in the *timed traces* of (s, a, s') or α, so as not to clutter the proof. However, it is clear that since the time-passage steps in $\mathcal{WD}^p$ are arbitrary, if we show *trace*(α) = *trace*(s, a, s') then *t-trace*(α) = *t-trace*(s, a, s'). We use u and u' to denote $R_{\text{TTWD}}(s)$ and $R_{\text{TTWD}}(s')$ respectively. We do not show the proof of correspondence for every action a of $\mathcal{TTCP}^h$ becasue some of proofs are very similar to the proofs of correspondence for similar actions of $\mathcal{TCP}^h$ presented in Chapter 7, and others are very similar to the proofs of some to the steps we do show. We focus on the steps that have proofs of correspondence that depend on features of specification $\mathcal{WD}$ that differ from specification D.

send-msg$_c$(open, m, close), a = passive-open, and *a = send-msg$_c$(open, m, close).*

The proof of correspondence for these steps is straightforward, and is similar to the proof of correspondence for the same steps in the proof of Lemma 7.1.

a = send-seg$_{cs}$(SYN, cc_send, sn$_c$, msg$_c$).

The proof of correspondence for this step is straightforward.

a = *receive-seg*$_{cs}$*(SYN, cc_send, sn*$_c$*, msg*$_c$*)*.

Let p be the segment received in this step, and let $cc\_send(p) = k$. We have two cases based on whether $s.mode_s$ = `listen` or not. If $s.mode_s \neq$ `listen` then a has no effect on the state of the server. Therefore, the corresonding timed execution fragment α of $\mathcal{WD}^p$ is (u, λ, u') the empty step. If $s.mode_s$ = `listen` then there are two subcases based on whether $k > cache\_cc$.

1. If $k > cache\_cc$ then α = *(u, choose-server-id(j), u‴, make-assoc(i,j), u″, drop*$_c$*(I, J, h, l), u′)* where both i and j are equal to k, and *u″, drop*$_c$*(I, J, h, l), u′)* represents a sequence of steps. The sequence of steps is of the form *u‴, drop*$_c$*(I, J, h, l), u″, drop*$_c$*(I', J', h', l'), ..., u')*. There is a *drop*$_c$ action for every h such that $h < k$, $(h, h) \notin s.assoc$, and $s.p\text{-}triple_c(h) \neq \emptyset$. For each h, the corresponding I = *dom(queue*$_{cs}(h)$, $J = \emptyset$, and $l :\in n$. The drop action is enabled for these queues because queues only contain one element, and the element is marked.

   Both α, and (s, a, s') have the empty trace. We need to show that α is enabled in state u. Since $s.mode_s$ = `listen`, we know that $u.choose\text{-}sid$ = `true`, so the *choose-server-id(j)* action is enabled in state u. By Invariant 10.8 we know that in $k \in s.used\text{-}id_c \cup \{s.crash\text{-}id_c\}$, so in the corresponding state u, $i \in u.used\text{-}id_c \cup \{u.crash\text{-}id_c\}$. After the *choose-server-id(j)* action we know $j \in u''.used\text{-}id_s$. From Invariant 10.11 we know that in state u'', $\forall\, x\ (i, x) \notin u''.assoc \wedge \forall\, y\ (y, j) \notin u''.assoc$. Therefore the *make-assoc(i,j)* action is enabled. We now need to show that u' is the correct corresponding state. For most variables it is clear that we get the correct correspondence. The interesting case is for $u'.queue_{cs}(k)$ if $msg(p) \neq$ `null`. Since by Invariant 10.9 we know that in state s, $(k, s.isn_s) \notin s.estb\text{-}cc \ \wedge\ (k, s.id_s) \notin s.assoc$, we know that $u.queue_{cs}(k)$ falls into either case (B) or (E) of the mapping to abstract queues. After step (s, a, s') the rule for the mapping changes from either (B) to (C) or (E) to (D). For either case, since α does not change the abstract queues, we need to show that $u.queue_{cs}(k) = u'.queue_{cs}(k)$, as defined by the R_{TTWD}. For the states where $u.queue_{cs}(k)$ fits into case (B), we know that $s.id_c = k$. Since $s.rcv\text{-}buf_s = \epsilon$, $s'.rcv\text{-}buf_s = s.rcv\text{-}buf_s \cdot msg(p)$, and $s.temp\text{-}data = s'.temp\text{-}data$ = `null`, to show that $u.queue_{cs}(k) = u'.queue_{cs}(k)$, we need to show that $msg(p) = s.cur\text{-}msg_c$ and that $s'.cur\text{-}msg_c = \epsilon$. Since after this step, we know $s'.ccsend = s'.cache\_cc$ and by Invariant 10.26 we know that $s.sn_c = sn(p)$; thus, $s'.sn_c < s'.ack_s$, so $s'.cur\text{-}msg_c = \epsilon$. Invariant 10.21 tells us that if $s.msg_c \neq$ `null` and $sn(p) = s.sn_c$, then $msg(p) = s.msg_c$.

For the states where $u.queue_{cs}(k)$ fits into case (E), after step (s, a, s'), since $(k, k) \in s'.assoc$, we know that $s'.p\text{-}triple_c(k) = \emptyset$, and since $s.rcv\text{-}buf_s = \epsilon$, $s'.rcv\text{-}buf_s = s.rcv\text{-}buf_s \cdot msg(p)$, and $s.temp\text{-}data = s'.temp\text{-}data =$ `null`, we clearly have $u.queue_{cs}(k) = u'.queue_{cs}(k)$ for this case. For $u.queue_{cs}(h)$ where $h < k$, that also fits into case (E), after (s, a, s'), $s'.p\text{-}triple_c(h) = \emptyset$. Since the single element in these queues is dropped after step α, the mapping is preserved.

2. If $k \leq cache\_cc$ then $\alpha =$ *(u, drop$_c$(I, J, h,l), u')* . For this case, after step (s, a, s'), $s'.mode_c =$ `syn-rcvd`, $cache\_cc = \infty$, and $s'.temp\text{-}data = msg(p)$. Again u'', *drop$_c$(I, J, h, l), u')* represents a sequence of steps. There is a *drop$_c$* action for every h such that $(h, h) \notin s.assoc$, and $s.p\text{-}triple_c(h) \neq \emptyset$. For each h, the corresponding I = *dom(queue$_{cs}$(h)*, $J = \emptyset$, and $l :\in n$. The drop action is enabled for these queues because the queues only contain one element, and that element is marked. If $msg(p) \neq$ `null`, then this change may affect the mapping to $u.queue_{cs}(i)$, where $s.id_c = i$. However, from Invariant 10.9 we know $(i, s.isn_s) \notin s.estb\text{-}cc$, so the mapping is not affected.

<u>*a = send-seg$_{sc}$(SYN, cc_rcvd, sn$_s$, ack$_s$)*</u>.

The proof of correspondence for this step is straightforward.

<u>*a = receive-seg$_{sc}$(SYN, cc_rcvd, sn$_s$, ack$_s$)*</u>.

For this step the the corresponding $\alpha = (u, \lambda, u')$. Let the received segment be p. If $s.mode_c \in \{$`syn-sent`, `syn-sent*`$\}$, $cc\_rcvd(p) = s.cc\_send$, and $ack(p) = s.sn_c + 1$, this step changes $mode_c$ to `estb` or `fin-wait-1`, and ack_c to $sn(p) + 1$. It also assigns msg_c to `null`, and adds $(id_c, sn(p))$ to *estb-cc*. These changes affect the mapping to $queue_{cs}(k)$, where $k = id_c$, because after the these changes, we may have case (C) of the mapping, where in state s, we have the first set of states of case (B). We only have the first set of states for case (B) because from Invariant 10.11 we know that $s.id_c$ is not yet part of an association pair nor is it part of an *estb-cc* pair. To show that the mapping is preserved after

α and a, we need to show that $u.queue_{cs}(k) = u'.queue_{cs}(k)$. Since $s'.msg_c = \texttt{null}$ we know $s'.cur\text{-}msg_c$ is empty, so to show $u.queue_{cs}(k) = u'.queue_{cs}(k)$ it is sufficient to show that $s'.rcv\text{-}buf_s$ is empty, and that if $s.temp\text{-}msg_c$ is not empty, then $s.msg_c = s'.temp\text{-}data$. If we do have case (C) of the mapping to $queue_{cs}(k)$ then by Invariant 10.14 we know $s'.mode_s = \texttt{syn-rcvd}$. If $s'.mode_s = \texttt{syn-rcvd}$, then Invariant 10.10 tells us that $s'.rcv\text{-}buf_s = \epsilon$. From Invariant 10.17 we know that $s.msg_c = s.temp\text{-}data$, and since *temp-data* does not change in this step, we know $s.msg_c = s'.temp\text{-}data$. Thus, $u.queue_{cs}(k) = u'.queue_{cs}(k)$.

$\underline{a = prepare\text{-}msg_c}$.

The proof of correspondence of actions can be shown in much the same way as was for the same action in T^h.

$\underline{a = prepare\text{-}msg_s}$.

The proof of correspondence of actions can be shown in much the same way as was for the same action in T^h.

$\underline{a = send\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)}$.

The proof of correspondence for this step is straightforward.

$\underline{a = receive\text{-}seg_{sc}(SYN,\ cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)}$.

Let p be the received segment, and let $cc\_rcvd(p) = k$. For this step we have two cases.

1. The first case occurs if $s.mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\}$, $s.msg_c \neq \texttt{null}$, and $cc\_rcvd(p) = s.cc\_send$, and state s is in the second set of states for case (B) of the mapping to $u.queue_{cs}(k)$. That is, if $(k, k) \in s.assoc \ \wedge \ (s.id_s \neq k \ \vee s.mode_s \in \{\texttt{rec}, \texttt{reset}\})$. For this case $\alpha = (u,\ drop_s(I,\ J,\ l,\ k),\ u')$, were $I = \emptyset$, $J = \{1\}$, and $l :\in \mathtt{N}$. Step (s, a, s') changes $mode_c$ to $\texttt{estb}$ or $\texttt{fin-wait-1}$, ack_c to $sn(p) + 1$, and msg_c to $\texttt{null}$. Thus, $cur\text{-}msg_c$ goes from being $(msg_c, \texttt{marked})$ to being the empty queue. However, since in α, the first element of $u.queue_{cs}(k)$ gets dropped, we get the right corresponding state. If $msg(p) \neq \texttt{null}$, then, $s'.rcv\text{-}buf_c = s.rcv\text{-}buf_c{\cdot}msg(p)$. This change may affect case (D) of the mapping to $u.queue_{sc}(k)$. Since this queue does not change in step α, we need to show that in state u' as defined by R_{TTWD}, $u'.queue_{sc}(k) = u.queue_{sc}(k)$. This is easy to see, because by definition of $s.p\text{-}triple_s$, $msg(p) = data(s.p\text{-}triple_s)$. Also by definition and Invariant 10.30, since $s'.mode_c \in$

$\{\texttt{estb}, \texttt{fin-wait-1}\}$ and $s'.ack_c = sn(p) + 1$, $s'.p\text{-}triple_s = \epsilon$. Thus, $u'.queue_{sc}(k) = u.queue_{sc}(k)$.

2. The second case is for all other states. For these states the corresponding $\alpha = (u, \lambda, u')$. Thus, if state s is not in the set of states of case one, then we know $s.id_s = k$. If $s.mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\}$, $cc\_rcvd(p) = s.cc\_send$, then this step changes $mode_c$ to `estb` or `fin-wait-1`, ack_c to $sn(p) + 1$, and msg_c to `null`, and if $msg(p) \neq$ `null` it gets concatenated to the end of $rcv\text{-}buf_c$. From Invariant 10.12 we know that for these states, $(k, k) \in s.assoc$. We also know that if we are not in the set of states for case one, then $s.id_s = k$. Therefore, we have case (C) of the mapping to $u.queue_{sc}(k)$. Invariant 10.26 tells us that $sn_s = sn(p)$. Therefore, the change of ack_s means $s.cur\text{-}msg_s$ is $(s.msg_s, \texttt{ok})$ and $s'.cur\text{-}msg_s$ is empty. Invariant 10.21 tells us that if $s.msg_s \neq$ `null` and $sn(p) = s.sn_s$, then $msg(p) = s.msg_s$. Therefore, since $s'.rcv\text{-}buf_s = s.rcv\text{-}buf_s \cdot msg(p)$, $u.queue_{sc}(k) = u'.queue_{sc}(k)$. The change of msg_c to `null` in this step may also affect the mapping to $u.queue_{cs}(k)$ for case (C), since it may affect $cur\text{-}msg_c$. However, for states other than the set of states of case one of this proof of correspondence, we know by Invariant 10.25 that $s.ack_s \geq ack(p)$. Therefore, since $ack(p) = s.sn_s + 1$, we know that $s.cur\text{-}msg_c = \epsilon$. Thus, the change of msg_c to `null` does not affect the mapping for this queue.

$\underline{a = send\text{-}seg_{cs}(cc\_send,\ sn_c,\ ack_c,\ msg_c)}$.

The proof of correspondence for this step is straightforward.

$\underline{a = receive\text{-}seg_{cs}(cc\_send,\ sn_c,\ ack_c,\ msg_c)}$.

Let p be the segment received in this action, and let $cc\_send(p) = k$. We have several cases.

1. The first case is if $s.mode_s = \texttt{rec} \vee (s.mode_s = \texttt{syn-rcvd} \wedge (cc\_send(p) \neq s.cc\_rcvd \vee s.ack_c \neq s.sn_s + 1)) \vee s.mode_s \in \{\texttt{closed}, \texttt{listen}\}$. For this case the corresponding α is the empty step. It is easy to see that we get the correct correspondence of states.

2. Case two occurs if $s.mode_s = \texttt{syn-rcvd}$, $ack(p) = s.sn_s + 1$, and $(k, k) \notin s.assoc$, then the corresponding α of $\mathcal{WD}^p$ is *(u, choose-server-id(j), u'', make-assoc(i,j), u')* where both i and j are equal to k. This case is similar to case two of of the step with a

$=$ *receive-seg*$_{cs}$*(SYN, cc_send,* sn_c*,* ack_c*,* msg_c*)*. From Invariant 10.10 we know that for this case $s.msg_s = \texttt{null}$, so $cur\text{-}msg_s$ is not affected by this step for this case. Thus, the interesting part of the proof of correspondence is show that $u.queue_{cs}(k) = u'.queue_{cs}(k)$, as defined by the R_{TTWD}. The mapping for this queue is affected if $s.temp\text{-}data \neq \texttt{null}$ and/or $sn(p) = s.ack_s$. If $s.temp\text{-}data \neq \texttt{null}$, it is concatenated to $s.rcv\text{-}buf_s$ and then assigned `null`. If $sn(p) = s.ack_s$ then $msg(p)$ is concatenated to $s.rcv\text{-}buf_s$ and ack_s is incremented. These changes affect the mapping of $u.queue_{cs}(k)$ for cases, (C) and (D). Cases (A), (B) and (E) do not apply here because we know by Invariant 10.12 that $(k, isn_s) \in estb\text{-}cc$.

For case (C), if $s.temp\text{-}data \neq \texttt{null}$ and $sn(p) \neq s.ack_s$, it is easy to see that the mapping is preserved, because from Invariant 10.10 we know that $s.rcv\text{-}buf_s = \epsilon$. Therefore, since for this stituation $s.temp\text{-}data$ is concatenated to $s.rcv\text{-}buf_s$, and then assigned `null`, we get the correct mapping. If $sn(p) = s.ack_s$, then since $s'.ack_s = s.ack_s + 1$, $s.cur\text{-}msg_c$ could go from being $(s.msg_c, \texttt{ok})$ to being empty. It is easy to see that $s.temp\text{-}msg$ is handled in the right way whether it is empty or not. If we are in case (C), then $s.id_c = k$. Invariant 10.26 tells us that $sn_c = sn(p)$. Therefore, the change of ack_s means $s.cur\text{-}msg_c$ is $(s.msg_c, \texttt{ok})$ and $s'.cur\text{-}msg_c$ is empty. Invariant 10.21 tells us that if $s.msg_c \neq \texttt{null}$ and $sn(p) = s.sn_c$, then $msg(p) = s.msg_c$. Therefore, since $s'.rcv\text{-}buf_s = s.rcv\text{-}buf_s \cdot msg(p)$, $u.queue_{cs}(k) = u'.queue_{cs}(k)$.

For case (D), Invariant 10.30 tells us that there are no other segments on the channel that has sequence number greater than $sn(p)$. Therefore, the change in ack_s means $s.p\text{-}triple_c(k)$ is $\{(k, msg(p), sn(p))\}$ and $s'.p\text{-}triple_c(k)$ is the empty set. However, as for case (C), since $s'.rcv\text{-}buf_s = s.rcv\text{-}buf_s \cdot msg(p)$ and Invariant 10.22 tells us that any segment with sequence number $sn(p)$ and connection count k must have the same message or the message is `null`. However, Invariant 10.23 tells us that any segment with sequence number $sn(p)$ has a message that is not `null`, so $u.queue_{cs}(k) = u'.queue_{cs}(k)$.

3. The third case is the same as the second case except with $(k, k) \in s.assoc$. For this case $\alpha =$ *(u, choose-server-id(j), u')*. The proof of correspondence is essentially the

same as for case two.

4. The fourth case occurs if $s.mode_s \in \{\texttt{last-ack}, \texttt{last-ack*}\}$ and $ack(p) = sn_s + 1$. We futher divide this case into two subcases.

   (a) The first subcase occurs if $(s.msg_s = \texttt{null} \vee (s.mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge s.id_c = s.id_s)$. This condition means that either $s.cur\text{-}msg_s = \epsilon$, or the second set of states for case (B) of the mapping to $u.queue_{sc}$ is not include in this subcase. The reason why be have the two subcases becomes clear when we discuss the second subcase, which is defined by the negation of the condition that defines this subcase. For this subcase α is *(u, set-nil$_s$, u$'$)*. Clearly a and α both have the empty trace. We must show that $set\text{-}nil_s$ is enabled in state u of $\mathcal{WD}^p$. Since $s.mode_s \in \{\texttt{last-ack}, \texttt{last-ack*}\}$, from our mapping we know $u.id_s \neq \texttt{nil}$, and from Invariant 10.15 we know that $u.mode_s = \texttt{inactive}$. The third part of the precondition requires that $\exists\ i$ s.t. $(i, u.id_s) \in\ u.assoc$. From Invariant 10.8 we know $s.id_s = s.cc\_rcvd$, and from Invariant 10.12 we know that since $s.mode_s \in \{\texttt{last-ack}, \texttt{last-ack*}\}$ and $k = cc\_rcvd$, then $(k, s.id_s) \in\ s.assoc$, so that part of the precondition holds for the corresponding state u.

   The fourth part of the precondition requires $u.queue_{cs}(k)$ to be empty. We only need to show this for cases (C) and (D) of the mapping to $u.queue_{cs}(k)$ because we know that $(k, s.id_s) \in\ s.assoc$, which rules out cases (A) and (E). From Invariant 10.12 we know from that if there exists j, such that $(k, j) \in s.estb\text{-}cc$, then $j = s.isn_s$, which along with the fact that $(k, s.id_s) \in\ s.assoc$ and $s.mode_s \in \{\texttt{last-ack}, \texttt{last-ack*}\}$, rules out case (B).

   We first examine case (C). Recall that the states for this case are states where $(s.id_c = k) \wedge (s.mode_c \notin \{\texttt{rec}, \texttt{reset}\}) \wedge ((k, s.isn_s) \in s.estb\text{-}cc \vee (k, s.id_s) \in s.assoc)) \wedge (s.mode_s \notin \{\texttt{rec}, \texttt{reset}\})$. To show that this queue is empty, we need to show that $s.send\text{-}buf_c$, $s.cur\text{-}msg_c$, $s.rcv\text{-}buf_s$, and $s.temp\text{-}msg$ are all empty. Invariant 10.10 tells us that $s.temp\text{-}msg$ is empty. If $s.mode_s = \texttt{last-ack}$, and $u.queue_{cs}(k)$ is defined for case (C) then Invariant 10.27 tells us that $s.mode_c \in \{$`syn-sent*`, `fin-wait-1`, `fin-wait-2`, `closing`, `timed-wait`,

$\texttt{last-ack}\}$, which coupled with Invariant 10.15 means $s.send\text{-}buf_c$ is empty. From Invariant 10.28 we know that that $s.sn_c < s.ack_s$, which means $s.cur\text{-}msg_c$ is empty. Finally, Invariant 10.31 indicates that $s.rcv\text{-}buf_s$ is empty. Therefore, $u.queue_{cs}(k)$ is empty.

Case (D) of the mapping to $u.queue_{cs}(i)$ occurs when $(s.id_c \neq k \vee s.mode_c \in \{\texttt{rec}, \texttt{reset}\}) \wedge ((k, s.isn_s) \in s.estb\text{-}cc \vee (k, s.id_s) \in s.assoc) \wedge (s.mode_s \notin \{\texttt{rec}, \texttt{reset}\})$. To show that this queue is empty, we need to show that $s.tmsg$, $s.p\text{-}triple_c(k)$ and $s.rcv\text{-}buf_s$ are empty. From Invariant 10.10 we know $s.temp\text{-}msg$ is empty. From Invariant 10.29 we that for all segments $q \in s.in\text{-}transit_{cs}$, $sn(q) < s.ack_s$ which means $s.p\text{-}triple_c(i)$ is empty, and from Invariant 10.32 we know that $s.rcv\text{-}buf_s$ is also empty.

The fifth and final part of the precondition for the $set\text{-}nil_s$ action in $\mathcal{WD}^p$ states that $(u.mode_c = \texttt{inactive} \vee u.id_c \neq i)$. From Invariant 10.33 we know this condition is true in state u.

After step (s, a, s'), $s'.mode_s = \texttt{closed}$, and after α, $u'.id_c = \texttt{nil}$. Therefore, the mapping is preserved for this variable. If $u.q\text{-}stat_{sc} = \texttt{live}$ and $u.id_c \neq i$ then $u'.queue_{sc}(u.id_s) = \epsilon$ and $u'.q\text{-}stat_{sc} = \texttt{dead}$. These values are the correct corresponding values as defined by R_{TTWD}. For this case, after step (s, a, s'), $s'.mode_s = \texttt{closed}$, $s'.ack_s$, $s'.msg_s$ and $s'.send\text{-}buf_s$ are all undefined. Since we know from Invariant 10.12 that $(s.id_s, s.id_s) \in s.assoc$, only cases (B) and (C) for the mapping to of $u.queue_{sc}(l)$, where if $s.id_s = l$ may be affected. However, condition $(s.msg_s = \texttt{null} \vee (s.mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge s.id_c = s.id_s)$ which we assume holds for this subcase means that if we have case (B) then $s.cur\text{-}msg_s = \epsilon$. After this step $u'.queue_{sc}(l)$ falls under case (A) which means it should be empty. From Invariant 10.15 we know $s.send\text{-}buf_s = \epsilon$, so making the buffer undefined in state s' does not affect the mapping to $queue_{sc}(l)$ for this case. Also since $s.cur\text{-}msg_s = \epsilon$ we know $u'.queue_{sc}(l)$ is empty. For case (C) of the mapping, after the step we have case (D). We know from Invariant 10.15 that $s.send\text{-}buf_s = \epsilon$. From Invariant 10.24 we know that $ack_c \geq ack(p)$, and since $ack(p) = s.sn_s + 1$, we know that for case (C) of the mapping $u.queue_{sc}(l)$,

$s.cur\text{-}msg_s = \epsilon$. Thus, in order to show that the mapping is preserved after (s, a, s') for this situation, we only need to show that $s'.p\text{-}triple_s$ is empty. Since $ack(p) = s.sn_s + 1$ and from Invariant 10.24 we know that $ack_c \geq ack(p)$ and from Invariant 10.7 we know that for all $p \in in\text{-}transit_{sc}$ $sn_s \geq sn(p)$, we know $s'.p\text{-}triple_s$ is the empty set, so the mapping to $queue_{sc}(l)$ is preseverved for this case.

(b) The second subcase occurs when $(s.msg_s \neq \texttt{null} \wedge (s.mode_c \in \{\texttt{rec}, \texttt{reset}\} \vee s.id_c \neq s.id_s)$. For this subcase α is *(u, drop$_s$(I, J, l,k), u'', set-nil$_s$, u')*. The proof of correspondence for this subcase is exactly the same as the proof of correspondence for the previous subcase, except in how we show that the mapping is preserved for case (B) for the mapping to $u.queue_{sc}(l)$. For case (B) the conditions for this subcase define a situation where $s.cur\text{-}msg_s = (s.msg_s, \texttt{marked})$. Thus, for this subcase $I = \emptyset$, $J = \{1\}$, and k is any arbitrary element of $\mathtt{N}$. We know the *drop$_s$(I, J, l, k)* action is enabled in state u, because $s.cur\text{-}msg_s$ corresponds to the first element of $u.queue_{sc}(l)$, and it is marked. From Invariant 10.15 we know $s.send\text{-}buf_s = \epsilon$, so after α, $u'.queue_{sc}(l)$ is empty, which is the correct state as defined by R_{TTWD}.

5. The fifth and final case is for all other states s. Like the previous case, we also divide this case into two subcases based on whether $ack(p) = s.sn_s + 1 \wedge s.cur\text{-}msg_s = (s.msg_s, \texttt{marked})$ or not. If $ack(p) = s.sn_s + 1$, we know by Invariant 10.24 that $s.ack_c > s.sn_s$. We also know by Invariant 10.13 that if $s.cc\_send = s.cc\_rcvd$ then $s.mode_c \notin \{\texttt{syn-sent}, \texttt{syn-sent*}\}$. Therefore, by definition, $s.cur\text{-}msg_s = \epsilon$, or $s.cur\text{-}msg_s = (s.msg_s, \texttt{marked})$.

   (a) For the first subcase $ack(p) \neq s.sn_s + 1 \vee s.cur\text{-}msg_s \neq (s.msg_s, \texttt{marked})$. For this subcase the corresponding $\alpha = (u, \lambda, u')$. For this case, ack_s and $rcv\text{-}buf_s$ may change as in case two, except we know from Invariant 10.10 that $temp\text{-}data = \texttt{null}$. Also $mode_s$ may change from `fin-wait-1` or `fin-wait1*` to `fin-wait-2`, or from `closing` or `closing*` to `timed-wait`. The proof that the mapping for $u.queue_{cs}(k)$ is preserved is the same as case two, and the possible changes to

$mode_s$ in $\mathcal{TTCP}^h$ do not affect its mapping to $mode_s$ in $\mathcal{WD}^p$.

(b) For the second subcase $ack(p) = s.sn_s + 1 \wedge\ s.cur\text{-}msg_s = (s.msg_s, \texttt{marked})$. For this subcase $\alpha = (u,\ drop_s(I,\ J,\ l,k),\ u')$. Here $I = \emptyset$, $J = \{1\}$, and k is any arbitrary element of $\mathbb{N}$. We know the $drop_s(I,\ J,\ l,\ k)$ action is enabled in state u, because $s.cur\text{-}msg_s$ corresponds to the first element of $u.queue_{sc}(l)$, and it is marked. The proof of correspondence is like the previous subcase, but for this subcase $u.queue_{sc}(l)$ is affected by the step since msg_s becomes $\texttt{null}$, which means $s'.cur\text{-}msg_s = \epsilon$. However, the fact that it is dropped by step α preserves the mapping.

$\underline{a = send\text{-}seg_{sc}(cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)}$.

The proof of correspondence for this step is straightforward.

$\underline{a = receive\text{-}seg_{sc}(cc\_rcvd,\ sn_s,\ ack_s,\ msg_s)}$.

Let p be the segment received in this action, and let $cc\_rcvd(p) = l$. This step is not quite symmetric to the step with $a = receive\text{-}seg_{cs}(cc\_send,\ sn_c,\ ack_c,\ msg_c)$, because the client is not assigned an id_c value in this step, nor is a pair added to *assoc* in this step. However, the effect on the mapping of the queues when a valid message is received, and when this segment causes the client to close, is basically symmetric to the situtations on the server side when the $receive\text{-}seg_{cs}(cc\_send,\ sn_c,\ ack_c,\ msg_c)$ action causes a valid message to be delivered or the server to close. For this step we break the proof of correspondence into two cases.

1. The first case occurs when $s.cc\_send = cc\_rcvd(p)$, $s.mode_c = \texttt{last-ack}$ and $ack(p) = sn_c + 1$. As for the symmetric step, we break this case into two subcases, base on where $s.cur\text{-}msg_c = (msg_c, \texttt{marked})$ or not.

   (a) For the subcase where $s.cur\text{-}msg_c \neq (msg_c, \texttt{marked})$, $\alpha = (u,\ set\text{-}nil_c,\ u')$. Clearly a and α both have the empty trace. We must show that $set\text{-}nil_c$ is enabled in state u of $\mathcal{WD}^p$. The only part of showing that this action is enabled in state u that is not symmetric to the case for the symmetric action is in showing that $u.queue_{sc}(l)$ is empty. We only have to show $u.queue_{sc}(l) = \epsilon$ for cases (C) and

(D). For case (C) we have to show that $s.send\text{-}buf_s$, $s.cur\text{-}msg_s$, and $s.rcv\text{-}buf_c$ are all empty. From Invariants 10.27 and 10.15 we know $s.send\text{-}buf_s$ is empty. From Invariant 10.28 we know that that $s.sn_s < s.ack_c$, which means $s.cur\text{-}msg_s$ is empty. Finally, Invariant 10.31 indicates that $s.rcv\text{-}buf_c$ is empty. Therefore, $u.queue_{sc}(l)$ is empty. For case (D) of the mapping to $u.queue_{sc}(l)$, we know from Invariant 10.29 that for all segments $q \in s.in\text{-}transit_{sc}$, $sn(q) < s.ack_c$ which means $s.p\text{-}triple_s$ is empty, and from Invariant 10.32 we know that $s.rcv\text{-}buf_c$ is also empty.

To show that after step (s, a, s') and α we get the correct corresponding states can be shown in a symmetric manner to the symmetric case, except that in showing $s.ack_s \geq ack(p)$ for $p \in in\text{-}transit_{sc}$, is not symmetric to showing $s.ack_c \geq ack(p)$ for $p \in in\text{-}transit_{cs}$. We need to show $s.ack_s \geq ack(p)$ for the set of states where case (C) of the mapping to $u.queue_{cs}(k)$ goes to case (D) after the step. For the set of states where we have case (C) of the mapping for this queue, Invariant 10.25 tells us that $s.ack_s \geq ack(p)$.

(b) For this second subcase $s.cur\text{-}msg_c = (msg_c, \texttt{marked})$. For this subcase $\alpha =$ *(u, $drop_c$(I, J, k, l), u″, set-nil$_c$, u′)*. Here $I = \emptyset$, $J = \{1\}$, and l is any arbitrary element of $\mathtt{N}$. The proof of correspondence for this subcase is the same as the previous subcase, except that *$drop_c$(I, J, k, l)* action ensures that we get the correct corresponding state for $u'.queue_{cs}(k)$.

2. The second case is for all other states. We divide these states into two subcases bases whether $cc\_rcvd(p) = s.cc\_send \land ack(p) = s.sn_c + 1 \land s.cur\text{-}msg_c = (s.msg_c, \texttt{marked})$ or not.

   (a) For the subcase where the condition is false, $\alpha = (u, \lambda, u')$. This subcase is interesting if $s.mode_c \notin \{\texttt{closed}, \texttt{syn-sent}, \texttt{syn-sent*}, \texttt{rec}, \texttt{reset}\}$, and $cc\_rcvd(p) = s.cc\_send \ \land sn(p) = s.ack_c$, because $msg(p)$ gets concatenated to $s.rcv\text{-}buf_c$ and ack_c gets incremented in this situation. These assignments affect the mapping for cases (C) and (D) of $u.queue_{sc}(l)$, where $l = cc\_rcvd(p)$, which we know by Invariant 10.8 is also equal to $s.id_c$. The other two cases are not affected because we know by Invariant 10.12 we know that $(l, l) \in s.assoc$.

For case (C), the fact that ack_c gets incremented may cause $s.cur\text{-}msg_s$ could go from being $(s.msg_s, \texttt{ok})$ to being empty. However, Invariant 10.26 tells us that $sn_c = sn(p)$. Therefore, the change of ack_c means $s.cur\text{-}msg_s$ is $(s.msg_s, \texttt{ok})$ and $s'.cur\text{-}msg_s$ is empty. Invariant 10.21 tells us that if $s.msg_s \neq \texttt{null}$ and $sn(p) = s.sn_s$, then $msg(p) = s.msg_s$. Therefore, since $s'.rcv\text{-}buf_c = s.rcv\text{-}buf_c \cdot msg(p)$, $u.queue_{sc}(l) = u'.queue_{sc}(l)$.

For case (D) of the mapping, Invariant 10.30 tells us that there are no other segments on the channel that has sequence number greater than $sn(p)$. Therefore, the change in ack_c means $s.p\text{-}triple_s$ is $\{(k, msg(p), sn(p))\}$ and $s'.p\text{-}triple_s(k)$ is the empty set. However, as for case (C), since $s'.rcv\text{-}buf_c = s.rcv\text{-}buf_c \cdot msg(p)$ and Invariant 10.22 tells us that any segment with sequence number $sn(p)$ and connection count l must have the same message or the message is null. However, Invariant 10.23 tells us that any segment with sequence number $sn(p)$ has a message that is not null, so $u.queue_{sc}(l) = u'.queue_{sc}(l)$.

(b) For the case where the condition is true, $\alpha = (u, drop_c(I, J, k, l), u')$. Here $I = \emptyset$, $J = \{1\}$, and l is any arbitrary element of $\mathtt{N}$. The proof of correspondence for this subcase is the same as the previous subcase, except that $drop_c(I, J, k, l)$ action ensures that we get the correct corresponding state for $u'.queue_{cs}(k)$.

$\underline{a = send\text{-}seg_{cs}(cc\_send, sn_c, ack_c, msg_c, FIN)}$.

The proof of correspondence for this step is straightforward.

$\underline{a = receive\text{-}seg_{cs}(cc\_send, sn_c, ack_c, msg_c, FIN)}$.

The proof of correspondence for this step is basically the same as cases one, two, three, and five of the step with $a = receive\text{-}seg_{cs}(cc\_send, sn_c, ack_c, msg_c)$.

$\underline{a = send\text{-}seg_{sc}(cc\_rcvd, sn_s, ack_s, msg_s, FIN)}$.

The proof of correspondence for this step is straightforward.

$\underline{a = receive\text{-}seg_{cs}(cc\_rcvd, sn_s, ack_s, msg_s, FIN)}$.

The proof of correspondence for this step is basically the same as cases one, two, and four of the step with $a = receive\text{-}seg_{cs}(cc\_rcvd, sn_s, ack_s, msg_s)$

a = *receive-msg*$_c$*(m)*.

The proof of correspondence for this step is basically the same as the proof of correspondence for the same step in $\mathcal{TCP}^h$.

a = *receive-msg*$_s$*(m)*.

The proof of correspondence for this step is basically the same as the proof of correspondence for the same step in $\mathcal{TCP}^h$.

a = *crash*$_c$.

The corresponding α in $\mathcal{WD}^p$ is the following sequence of steps *(u, crash*$_c$*, u''', mark*$_c$*(I, J, j), u'', drop*$_c(I', J', k, l)$*, u')*. Clearly, α has the same trace as a since *crash*$_c$ is the only external action in the sequence.

First we show that this sequence of steps is enabled in $\mathcal{WD}^p$. After *crash*$_c$, *rec*$_c$ is `true`, so *mark*$_c$*(I, J, j)* is enabled, and *drop*$_c(I', J', k, l)$ is enabled if I', J', k and l are defined correctly. We define I, J, I', J', j, k, and l below and show that $R_{\text{TTWD}}(s')$ is indeed the state u' we get after the sequence of steps α.

The changes in state caused by step (s, a, s') is that $s'.mode_c$ = `rec`, $s'.crash\text{-}id_c = s.id_c$, and $s'.used\text{-}id_c = s.used\text{-}id_c \setminus s.id_c$. After α in $\mathcal{WD}^p$, we have $u'.mode_c$ = `rec`, $u'.crash\text{-}id_c = u.id_c$, $u'.used\text{-}id_c = u.used\text{-}id_c \setminus u.id_c$. It is clear that the mapping is preserved for $u'.mode_c, u'.used\text{-}id_c$, and $u'.crash\text{-}id_c$. The interesting part of the proof of correspondence lies in showing the mapping is preserved for $u'.queue_{cs}(i)$ and for $u'.queue_{sc}(j)$ if there exists j such that $(u.id_c, j) \in u.assoc$ and $u.q\text{-}stat_{sc}(j)$ = `live`. If there is no such j, then $j \in$ `N`, J and J' is equal to the empty set, and $l \in$ `N`. Now assume there is such a j. We examine the correspondence of $u'.queue_{cs}(u.id_c)$ and $u'.queue_{sc}(j)$ separately. We can separate the examination of these variables because effect of the *mark*$_c$*(I, J, j)* and *drop*$_c(I', J', k, l)$ actions on the queues are independent.

We start with $u'.queue_{cs}(i)$. For this variable the proof of correspondence is similar to the proof present in Chapter 7 for this variable and this step in the proof of Lemma 7.1.

1. The first case is for cases (A), (D), and (E) of the mapping to $u.queue_{cs}(i)$. For these cases $I = I' = \emptyset$ and $k = i$, so α does not change $u.queue_{cs}(i)$. The correspondence of states is preserved because step (s, a, s') does not affect the mapping for these queues.

2. The second case is for case (B) of the mapping to $u.queue_{cs}(i)$ if $((k,j) \in s.estb\text{-}cc \wedge (s.isn_s \neq j \vee s.mode_s \in \{\texttt{rec}, \texttt{reset}\})) \vee ((k,k) \in s.assoc \wedge (s.id_s \neq k \vee s.mode_s \in \{\texttt{rec}, \texttt{reset}\}))$. For this case, after step (s, a, s'), $u'.queue_{cs}(i)$ should be the empty set. Thus, for this case $I = I' = dom(u.queue_{cs}(i))$, and $k = i$. We clearly get the correct correspondence of states.

3. We now examine case (C) of the mapping to $u.queue_{cs}(i)$. If $u.queue_{cs}(i)$ falls under case (C), after step (s, a, s'), $u'.queue_{cs}(i)$ falls under case (D). We can break this case into two subcases based on whether $s'.p\text{-}triple_c(i)$ is empty or not. For both subcases $i = k$. We use the following preliminary definition: $suffix_{rb} = \{i | |s.rcv\text{-}buf_s| < i \leq maxindex(u.queue_{cs}(i))\}$. That is, $suffix_{rb}$ is the suffix of $u.queue_{cs}(i)$ that starts with the element that maps to the first element after $s.rcv\text{-}buf_s$.

   (a) If there exists a segment $p \in s.in\text{-}transit_{cs}$, such that after the $crash_c$ action, $s'.p\text{-}triple_c(i) \neq \emptyset$. Therefore, $I = suffix_{rb}$ and $I' = suffix_{rb} / maxindex(suffix_{rb})$. I' is the suffix of $u.queue_{cs}(i)$ that starts with the element that maps to the second element after $s.rcv\text{-}buf_s$ which is also the first element after $s'.p\text{-}triple_c(i)$. After a, we have case (D) of the mapping to $queue_{cs}(i)$, but since α deletes all the elements after $s'.p\text{-}triple_c(i)$, we get the right corresponding state.

   (b) Case two occurs for all other states for case (C). That is, states where $s'.p\text{-}triple_c = \emptyset$. For this case $I = I' = suffix_{rb}$. After α $u'.queue_{cs}(i)$ corresponds to the $s'.rcv\text{-}buf_s$. However, this still satisfies the mapping of $u'.queue_{cs}(i)$ for case (D) because $s'.p\text{-}triple_c(i)$ is empty.

Now we examine the case for $u'.queue_{sc}(j)$, where $(u.id_c, j) \in u.assoc$ and $u.q\text{-}stat_{sc}(j) = \texttt{live}$. For this variable the mapping is affected if we have case (C) or (D) for $u'.queue_{sc}(j)$.

1. If we have case (D), then after step (s, a, s'), $u'.queue_{sc}(j)$ should be empty. For this case let $J = J' = dom(u'''.queue_{sc}(j))$ and let $l = j$. Therefore, after α, $u'.queue_{sc}(j) = \epsilon$.

2. If $u.queue_{sc}(j)$ is in case (C) of the mapping, then after step (s, a, s'), $u'.queue_{sc}(j)$, is in case (B). We use the following preliminary definition: $prefix_{rb} = \{i | 1 \geq i \leq$

$|s.\mathit{rcv\text{-}buf}_c|\}$. That is, prefix_{rb} is the prefix of $u.\mathit{queue}_{sc}(j)$ that consists of the elements of $s.\mathit{rcv\text{-}buf}_c$. For this case we have two subcases.

(a) The first subcase occurs if $s.\mathit{cur\text{-}msg}_s = \epsilon$ and $s'.\mathit{cur\text{-}msg}_s = (s'.\mathit{msgs}_s, \mathtt{ok})$. We know that if $u.\mathit{last\text{-}msg}_c \neq \mathtt{null}$ it is added to the front of $u.\mathit{queue}_{sc}(j)$ in the *(u, crash$_c$, u''')* step of α. Thus, if $u.\mathit{last\text{-}msg}_c \neq \mathtt{null}$, $u'''.\mathit{queue}_{sc}(j)$ is the concatenation of $s.\mathit{last\text{-}msg}_c$, $(s.\mathit{rcv\text{-}buf}_s \times \mathtt{ok})$, and $(s.\mathit{send\text{-}buf}_s \times \mathtt{ok})$, since $s.\mathit{cur\text{-}msg}_s = \epsilon$. For this case $J = J' = \mathit{prefix}_{rb}$. If $s.\mathit{rcv\text{-}buf}_c = \epsilon$, then $J = \emptyset$, so no element of $u'''.\mathit{queue}_{sc}(j)$ gets marked or dropped. Since by mapping R_{TTWD} $u'.\mathit{queue}_{sc}(j)$ is the concatenation of $s'.\mathit{cur\text{-}msg}_s$ and $(s'.\mathit{send\text{-}buf}_s \times \mathtt{ok})$, we need to show that $s'.\mathit{cur\text{-}msg}_s = s.\mathit{last\text{-}msg}_c$. We know this is true by Invariant 10.18. If $s.\mathit{rcv\text{-}buf}_c \neq \epsilon$, then because an extra element is added to the front of the queue in this situation where $u.\mathit{last\text{-}msg}_c \neq \mathtt{null}$, prefix_{rb} is the prefix of $u'''.\mathit{queue}_{sc}(j)$ up to, but not including the last element of $s.\mathit{rcv\text{-}buf}_c$. In order to show that the mapping is preserved for this scenario, we must show that the last element of $s.\mathit{rcv\text{-}buf}_c$ is equal to $s'.\mathit{cur\text{-}msg}_s$. Invariant 10.19 tells us that this is true.

    If $u.\mathit{last\text{-}msg}_c = \mathtt{null}$, then no element is added to the front of $u.\mathit{queue}_{sc}(j)$ in the *(u, crash$_c$, u''')* step of α. Therefore, $u'''.\mathit{queue}_{sc}(j)$ is the concatenation of $(s.\mathit{rcv\text{-}buf}_s \times \mathtt{ok})$, and $(s.\mathit{send\text{-}buf}_s \times \mathtt{ok})$. For this case we know that $s.\mathit{rcv\text{-}buf}_c \neq \epsilon$, because of Invariant 10.20. For this situation $J = J' = \mathit{prefix}_{rb} \setminus \mathit{maxindex}(\mathit{prefix}_{rb})$, that is J is the prefix of $u'''.\mathit{queue}_{sc}(j)$ up to, but not including the last element of $s.\mathit{rcv\text{-}buf}_c$. Again by Invariant 10.19 we know that this element is the same as $s'.\mathit{cur\text{-}msg}_s$.

(b) The second subcase occurs if $s.\mathit{cur\text{-}msg}_s \neq \epsilon$, or $s'.\mathit{cur\text{-}msg}_s = \epsilon$. In the case where $s'.\mathit{cur\text{-}msg}_s = \epsilon$, we know that $s.\mathit{cur\text{-}msg}_s$ also equals ϵ, because step (s, a, s') cannot make $\mathit{cur\text{-}msg}_s$ go from not being empty to being empty. Therefore, for this subcase, whether $s.\mathit{cur\text{-}msg}_s \neq \epsilon$, or $s'.\mathit{cur\text{-}msg}_s = \epsilon$. $\mathit{cur\text{-}msg}_s$ does not change after step (s, a, s'), so we need to delete all the elements that are ahead of $s.\mathit{cur\text{-}msg}_s$ in the abstract queue. Therefore, if $u.\mathit{last\text{-}msg}_c \neq \mathtt{null}$, $J = J' = \mathit{prefix}_{rb} \cup \{\mathit{maxindex}(\mathit{prefix}_{rb}) + 1\}$. That is, J includes $u.\mathit{last\text{-}msg}_c$ and all of $s.\mathit{rcv\text{-}buf}_c$. If $u.\mathit{last\text{-}msg}_c = \mathtt{null}$, then $J = J' = \mathit{prefix}_{rb}$.

$\underline{a = crash_s.}$

The proof of correspondence for this step is almost symmetric to the case for $a = crash_c$. The corresponding $\alpha = (u,\ crash_s,\ u''',\ mark_s(I,\ J,\ j),\ u'',\ drop_s(I', J', k, l),\ u')$ is symmetric, and the proof of correspondence for $u'.queue_{sc}(j)$ where $j = u.id_s$ is essentially symmetric to the case for $u'.queue_{cs}(i)$ where $i = u.id_c$ for the $a = crash_c$ step. It is in the proof of correspondence for $u'.queue_{cs}(i)$, for this step, where the non-symmetry occurs. The mapping for this queue is affected if $u.queue_{cs}(i)$ falls under case (C) or (D) for the mapping of this variable. The non-symmetry comes from the fact that in the mapping for these two cases, includes the *temp-msg* derived variable and there is no symmetric counterpart for $u.queue_{sc}(j)$. However, the proof of correspondence proceeds in much the same manner. Thus, we have two cases.

1. If we have case (D), then after step (s, a, s'), $u'.queue_{cs}(i)$ should be empty. For this case let $J = J' = dom(u'''.queue_{sc}(i))$ and let $l = i$. Therefore, after α, $u'.queue_{cs}(i) = \epsilon$.

2. If $u.queue_{cs}(i)$ is in case (C) of the mapping, then after step (s, a, s'), $u'.queue_{cs}(i)$, is in case (B). We can break this case into subcases based on whether $s.temp\text{-}msg = \epsilon$ or not. If $s.temp\text{-}msg = \epsilon$, then the proof of correspondence is symmetric to the case for $u'.queue_{sc}(j)$ of the step with $a = crash_c$. Therefore, we only show the proof of correspondence for the case where $s.temp\text{-}msg \neq \epsilon$. From Invariant 10.10 we know that if $s.temp\text{-}msg \neq \epsilon$ then $s.last\text{-}msg_s = \texttt{null}$, so we do not have to worry about $u.last\text{-}msg_s$ getting added to the front of $u.queue_{cs}(i)$. We have two subcases.

   (a) The first subcase occurs if $s.cur\text{-}msg_c = \epsilon$ and $s'.cur\text{-}msg_c = (s'.msg_c, \texttt{marked})$. For this case $J = \{1\}$ and $J' = \emptyset$, we know we get the right corresponding state because Invariant 10.16 tells us that in this situation $s.temp\text{-}msg = s'.cur\text{-}msg_c$.

   (b) The second subcase occurs if $s.cur\text{-}msg_c \neq \epsilon$, or $s'.cur\text{-}msg_c = \epsilon$. Since step (s, a, s') cannot make $cur\text{-}msg_c$ go from not being empty to being empty, for this subcase $cur\text{-}msg_c$ does not change. Thus, we need to delete $s.temp\text{-}msg$ from the front of the queue, so $J = J' = \{1\}$. This clearly gives the correct corresponding state.

$\underline{a = recover_c.}$

The proof of correspondence for this step is also very similar proof of correspondence present in Chapter 7 for this this step of $\mathcal{TCP}^h$ in the proof of Lemma 7.1

The corresponding α of $\mathcal{WD}^p$ is *(u, mark$_c$(I, J, j), u''', drop$_c$(I,J,k,l), u'', recover$_c$, u')*. Since only $recover_c$ is external, the traces of a and α are clearly the same. We first show that this sequence of steps is enabled in $\mathcal{WD}^p$. The action $recover_c$ is enabled in $\mathcal{TTCP}^h$ if $s.mode_c$ = `rec`. This state maps to $u.rec$ = `true` in which case *mark$_c$(I, J, j)* is enabled, and *drop$_c$(I,J,k,l)* is also enabled. Since neither *mark$_c$(I, J, j)* nor *drop$_c$(I,J,k,l)* changes $u.rec$, then $recover_c$ is also enabled. We define I, J, k, l below.

After step (s, a, s'), $s'.mode_c$ = `closed` and $s'.cache\_cc = \infty$. This change affects the mapping for $u.rec_c$, $u.id_c$, $u.queue_{cs}(i)$, and $u.q\text{-}stat_{cs}(i)$, where $i = s.id_c$. After step α, $u'.rec_c$ = `false` and $u'.id_c$ = `nil`, so the mapping is preserved for those variables. Since $u.queue_{sc}(j)$ is not affected by this step, $J = \emptyset$ and j and k are any arbitrary values in N. For $u.queue_{cs}(i)$ the mapping is only affected by step (s, a, s') if state s is in case (B) of the mapping to $queue_{cs}(i)$, because case (C) does not hold if $s.mode_c$ = `rec`, and for cases (A), (D), and (E) the action does not affect the mapping. Therefore, $u.queue_{cs}(i)$ for cases (A), (D), and (E) $I = \emptyset$ and $k = i$.

For case (B) of the mapping to $u.queue_{cs}(i)$, after step (s, a, s') it is in group (A). Let I = *dom(u.queue$_{cs}$(i))* and $k = i$. The mapping is preserved because after a, $s.send\text{-}buf_c$ is deleted and $s'.cur\text{-}msg_c$ is empty. Finally, to show that the mapping for $u.q\text{-}stat_{cs}(i)$ is preserved we note the mapping for this variable is affected for two cases. The first case is if $i \notin s.estb\text{-}cc \ \wedge (i, i) \notin s.assoc$ then because $s.id_c = i$, $u.q\text{-}stat_{cs}(i)$ is `live`. After step (s, a, s'), $u'.q\text{-}stat_{cs}(i)$ should be `dead` since $s'.id_c$ = `nil`. For this case $u.queue_{cs}(i)$ is in case (B) of the mapping to abstract queues, so after the mark and drop actions this queue is empty. In the $(u'', recover_c, u')$ step of α, $u''.queue_{cs}(i)$ is empty and for all j, $(u''.id_c, j) \notin u''.assoc$, then $u'.q\text{-}stat_{cs}(i)$ = `dead`, so we get the correct corresponding state. The other case where $u.q\text{-}stat_{cs}(i)$ is affected is if $(i, i) \in s.assoc \ \wedge s.id_s \neq i$. Again this is a case where $u.q\text{-}stat_{cs}(i)$ = `live` and $u'.q\text{-}stat_{cs}(i)$ should be `dead`. This is another case where $u.queue_{cs}(i)$ is in case (B) of the mapping to abstract queues, and again after the $(u'', recover_c, u')$ step of α, $u'.q\text{-}stat_{cs}(i)$ = `dead`. Therefore, we have the correct correspondence of states.

<u>a = *recover*$_s$.</u>

This step is essentially symmetric to $a = recover_c$. Except that because after step (s, a, s') $cache\_cc = \infty$, this steps affects the mapping for case (E) of $u.queue_{cs}(h)$. Thus, for this step α = *(u, mark$_s$(I, J, j), u'''', drop$_s$(I,J,k,l), u''', drop$_c$(I', J', h, l), u'', recover$_s$, u')*. Here again u'', $drop_c(I', J', h, l), u')$ represents a sequence of steps. There is a $drop_c$ action for every h such that $(h, h) \notin s.assoc$, and $s.p\text{-}triple_c(h) \neq \emptyset$. For each h, the corresponding $I' = dom(queue_{cs}(h)$, $J' = \emptyset$, and $l :\in n$. The drop action is enabled for $u.queue_{cs}(h)$ because the queues only contain one element, and the element is marked. After (s, a, s'), $s'.p\text{-}triple_c(h) = \emptyset$. Since the single element in these queues is dropped after step α, the mapping is preserved.

<u>a = *timeout*$_c$.</u>

The proof of correspondence for this step is basically the same as case one of the proof of correspondence for the step with a = *receive-seg$_{sc}$(cc_rcvd, sn$_s$, ack$_s$, msg$_s$)*.

<u>a = *timeout*$_s$.</u>

The proof of correspondence for this step is basically the same as case four of the proof of correspondence for the step with a = *receive-seg$_{sc}$(cc_send, sn$_c$, ack$_c$, msg$_c$)*.

<u>$a = drop_{cs}(p)$ and $a = drop_{sc}(p)$</u> (from $Ch_{cs}(\mathcal{P})$ and $Ch_{sc}(\mathcal{P})$ respectively).

The proof of correspondence for these step is basically the same as the proof of correspondence for the same step in $\mathcal{TCP}^h$.

<u>$a = duplicate_{cs}(p)$ and $a = duplicate_{sc}(p)$</u> (from the $Ch_{cs}(\mathcal{P})$ component $Ch_{sc}(\mathcal{P})$ respectively).

The proof of correspondence for these steps is basically the same as the proof of correspondence for the same step in $\mathcal{TCP}^h$.

<u>$a = \nu(t)$ *(time-passage)*</u>

The corresponding α in $\mathcal{WD}^p$ is $(u, \nu(t), u')$, the time-passage action of the patient $\mathcal{WD}^p$.

<u>a = *send-seg$_{sc}$(RST, ack$_s$, rst-seq$_s$)*.</u>

The proof of correspondence for this step is straightforward.

$a = \mathit{receive\text{-}seg}_{sc}(RST,\ ack_s,\ rst\text{-}seq_s)$.

The proof of correspondence for this step is basically the same as the proof of correspondence for the $a = \mathit{crash}_c$ step.

$a = \mathit{send\text{-}seg}_{cs}(RST,\ ack_c,\ rst\text{-}seq_c)$.

The proof of correspondence for this step is straightforward.

$a = \mathit{receive\text{-}seg}_{cs}(RST,\ ack_c,\ rst\text{-}seq_c)$.

The proof of correspondence for this step is basically the same as the proof of correspondence for the $a = \mathit{crash}_s$ step.

$a = \mathit{shut\text{-}down}_c$.

The proof of correspondence for this step is basically the same as the proof of correspondence for the $a = \mathit{recover}_c$ step.

$a = \mathit{shut\text{-}down}_s$.

The proof of correspondence for this step is basically the same as the proof of correspondence for the $a = \mathit{recover}_s$ step.

This concludes the simulation proof. ■

10.4.3 Proof of trace inclusion

We can now proof that the GTA model of T/TCP, $\mathcal{TTCP}$, implements a patient version of Specification $\mathcal{WS}$.

Theorem 10.2

$\mathcal{TTCP} \sqsubseteq_t \mathit{patient}(\mathcal{WS})$.

Proof: From Lemma 10.3 we get that $\mathcal{TTCP}^h \leq_R^t \mathcal{WD}^p$, which because of the soundness of timed refinement mapping (Theorem 3.6) and the soundness of adding history variables (Theorem 3.9) implies that $\mathcal{TTCP} \sqsubseteq_t \mathcal{WD}^p$. From Theorem 10.1 we know $\mathcal{WD} \sqsubseteq \mathcal{WS}$. Using *Embedding Theorem* of [31] presented in Chapter 3 we now get $\mathcal{WD}^p \sqsubseteq_t \mathit{patient}(\mathcal{WS})$. Thus, we now have $\mathcal{TTCP} \sqsubseteq_t \mathcal{WD}^p$ and $\mathcal{WD}^p \sqsubseteq_t \mathit{patient}(\mathcal{WS})$. Therefore, since the subset relation and thus the implements relation is transitive we get $\mathcal{TTCP} \sqsubseteq_t \mathit{patient}(\mathcal{WS})$. ■

Chapter 11

An Impossibility Result

11.1 Introduction

The duplicate delivery in T/TCP (presented in Chapter 9) occurs because the TAO mechanism bypasses the three-way handshake protocol in an effort to achieve efficient transactions. The observation the the TAO mechanism may cause duplicate delivery lead us to consider whether any protocol could deliver streams of data reliablely and still have fast transactions, and under what conditions.

Shankar and Lee [33] show that some timing assumptions are needed for T/TCP and protocols that work in the same general manner to provide fast transactions and still deliver data without duplication. They assume that the protocols use counters. In this chapter we prove that if the hosts do not have "accurate" clocks it is impossible for any protocol to satisfy our specification and still provide "fast" transactions. We elaborate on what we mean by "accurate" clocks, "fast" transactions later in the chapter. In the proof of the impossibility result, the hosts are allowed to have infinite and stable sets of unique identifiers (UID's), but not counters.

T/TCP is designed to be a reliable transport lever protocol that also support efficient transactions. An efficient transaction is one that is completed in round trip time (RTT) plus server processing time (SPT), where RTT is the the time it takes for a packet to make a round trip across the network and SPT is the time the server takes to process the request and produce a response. In order for a transaction to be completed in this amount of time, it is necessary that the client can send a message to the server that can be accepted using only one trip across the network.

In typical network situations, client and server hosts may have several different connections in parallel. Additionally, there may be different *incarnations* of the same connection, as the connection is opened closed and then opened again. In order to ensure reliable delivery, hosts maintain some state information for each incarnation of a connection. However, because of the number of connections a host may be involved with, this state information cannot be maintained forever. Therefore, hosts will periodically quiesce, that is, delete state information associated with a connection. Kleinberg, Attiya, and Lynch in [17] prove tradeoffs between quiesce time and message delivery. They prove that in the absence of crashes, in an asynchronous setting where the client and server both have an infinite set of unique identifiers (UID's) and must quiesce, a three-way handshake is necessary to guarantee reliable message delivery. In an environment where there are crashes, Kleinberg et al. [17] show that even in a system with synchronized clocks, if the server does not remember the time of the last crash, then a three-way handshake is necessary for reliable message delivery.

Another approach to the design of reliable transport level protocol is to use *timer-based* mechanisms. For example, the Delta-t protocol [37] relies on clocks that run at the rate of real time and exploits the knowledge of the maximum segment lifetime (MSL). In this type of environment, Kleinberg et al. show that either it takes a three-way handshake to deliver a message, or at least the maximum packet lifetime must elapse before quiescence. If the MSL is unknown, then they show that the three-way handshake is required. If the client and server hosts are assumed to have approximately synchronized clocks, then the protocol by Liskov, Shrira, and Wroclawski [19] only requires one trip across the network for the server to deliver a message from the client, and quiesce time depends on the message delivery time.

Braden and Clark [8, 7], as we have seen, takes a different approach in there design of T/TCP. Their approach does not rely on approximately synchronized clocks or strict enforcement of MSL. Their approach is based on the idea that some information related to incarnations can be stored indefinitely and efficiently in caches when a connection closes, and that the protocol while ensuring efficient transactions most of the time (when the caches have the appropriate state), is allowed to be inefficient in some situations — typically after crashes. T/TCP uses counters, but we know form the work of Søgaard-Andersen et al. [35], that reliable at-most-once delivery can be achieved using just infinite and stable sets of UID's. We also know from the work of Shankar and Lee [33], that some timing information is needed inorder for protocols like T/TCP to have fast transactions and reliable delivery. We know that in T/TCP if a message from the client is successfully delivered by the server and there has not been a crash since the delivery of this message, then the state of the caches are appropriated. Therefore, we can weaken the preformance criteria for protocols that allow efficient transactions and reliable data delivery to require the following. If a message from the client user is successfully delivered by the server and there has not been a crash since the delivery of this message, then the next message from the client to the server should be delivered in one trip across the network. We want this performance only if the clocks of the client and server are running at the rate of real time. However, since protocols that rely only on the fact that the client and server have UID's can guarantee at-most-once message delivery, we want the at-most-once delivery property to hold even when the timing assumptions do not hold.

We formally state the properties of the system the protocol should work in and the properties we want to protocol to exhibit in the next section.

11.2 The underlying formal model

In this section we present the underlying formal model used for the impossibility result in this chapter. We start with the general timed automaton (GTA) model presented in Chapter 2 and make several additions until we get the model we need.

11.2.1 The clock GTA model

In the system we want to model, the client and server have access to local clocks, but are not able to use real time. However, a GTA A does have access to real time. To get this "local clock" property, we use the *clock general timed automaton (CGTA)* model of [29]. A CGTA, A, is a GTA with a special variable $clock_A$ (or just $clock$ if A is clear from the context) that has type $R^{\geq 0}$ and is the local time of that automaton. A CGTA A has the following three axioms.

1. $clock_A$ changes only with time passage actions $(\nu(t), t \in R^+)$.

2. $clock_A$ is monotonically non-decreasing.

3. If $(s, \nu(t), s')$ is a step then $\forall\ t' > 0, (s, \nu(t'), s')$ is also a step.

The $clock_A$ variable is used to model local time which may not be the same as real time. However, we want $clock_A$ to be like real time in that it only changes when time changes and it does not go backwards. The first two axioms capture these properties. Since $clock_A$ is supposed to represent the local time of a process, real time should not affect the actions of the process in any manner. This property is captured by the third axiom. We also refer to this property as *real time independence*.

11.2.2 Clock functions

Given a CGTA A, we may want to specify the values that $clock_A$ takes on for a timed execution fragment. To specify the values we introduce *clock functions*. These functions take real time as input and return values for the $clock_A$ variable of a CGTA A. A clock function $cf : R^{\geq 0} \rightarrow R$ has the following properties:

1. It is monotonically non-decreasing.

2. It is unbounded.

We use clock functions and the *fix-clock* operator to fix values of clock variables relative to real time for a timed execution fragment of a CGTA. We define *fix-clock* below.

Definition 11.1 (fix-clock)

Let A be a CGTA and cf be a clock function such that $\exists\ s \in start(A)$ such that $s.clock = cf(0)$. Then define *fix-clock(A, cf)*, denoted as A_{cf}, to be the CGTA with a *now* variable that gives real time, such that:

1. $states(A_{cf}) = \{(s, u) \mid u \in R^{\geq 0}\ \wedge s.clock = cf(u)\}$.

2. $start(A_{cf}) = \{(s,u) \in states(A_{cf}) \mid s \in start(A) \wedge u = 0\}$.

3. $(in(A_{cf}),\ out(A_{cf}),\ int(A_{cf})) = (in(A), out(A),\ int(A))$.

4. $steps(A_{cf})$ consists of the steps:

   (a) $\{((s,u),\pi,(s',u)) \mid (s,\pi,s') \in steps(A) \wedge ((s,u) \text{ and } (s',u) \in states(A_{cf})) \wedge \pi \in disc(A)\}$

   (b) $\{((s,u),\nu(t),(s',u{+}t)) \mid (s,\nu(t),s') \in steps(A) \wedge ((s,u) \text{ and } (s',u{+}t) \in states(A_{cf}))\}$.

   □

The states of A_{cf} are obtained by pairing each state s of A with the real time such that the clock function applied to that real time is the value of *clock* in state s. The set of start states of A_{cf} is the set of states of A_{cf} such that the first component of each state is an element of the set of start states of A and the second component is 0. In order for A_{cf} to be a GTA, $start(A_{cf})$ must be non-empty and real time must be 0 for every element of the set of start states. That is why we have the restriction that the fix-clock operator can only be applied to a CGTA A and a clock function cf if there exists a start state of A such that the value of *clock* in that start state is equal to $cf(0)$. The third component of A_{cf}, the partition of the actions, is the same for A_{cf} as it is for A. The final component of A_{cf}, the set of steps, can be partitioned into two sets — steps that have discrete actions and steps that have time-passage actions. The steps that have a discrete action π are obtained by taking any step of the form $(s,\pi,s') \in steps(A)$, and having a transition, via π, from states of A_{cf} that have s as the first component, to states of A_{cf} that have s' as the first component. The second component of the pairs remains the same for both states of A_{cf} because discrete actions cannot change the *clock* variable of A. For time-passage actions, the second component of the states of A_{cf} must change to reflect the length of time passage.

11.2.3 Liveness

The general timed automaton model is useful for proving safety properties and some liveness properties. However, for the impossibility result we prove, we need more general liveness properties than can be handled by the GTA model. In particular, we want the automaton to not block time. To get this property we use a model defined in [31, 35]. We call it the *live GTA* model[1] because its first component is a GTA. Before we define the live GTA model we need some preliminary definitions. The reader is referred to [31] for more complete definitions and more discussions of the model.

11.2.4 Timed executions

Recall that in Chapter 2 *finite, admissible, Zeno*, and all timed executions of a GTA A, denoted by $t\text{-}exec^*(A)$, $t\text{-}exec^\omega(A)$, $t\text{-}exec^Z(A)$, and *t-exec(A)* respectively, are defined. Also defined in that chapter are *finite, admissible, Zeno*, and all timed execution fragments denoted by $t\text{-}frag^*(A)$, $t\text{-}frag^\omega(A)$, $t\text{-}frag^Z(A)$, and *t-frag(A)*. We are particularly concerned with *Zeno* timed execution fragments because these can block time. Intuitively, Zeno timed executions can block time in two ways. The first occurs if there are infinitely many occurrences of non-time-passage actions, but for which there is a finite upper bound on the last time of the execution fragment. The second occurs if there are finitely many occurrences of non-time-passages actions and infinitely many time passage actions, but with a finite upper bound on the last time of the execution fragment.

11.2.5 Live GTA

Since the GTA model allows *Zeno* timed executions, *liveness conditions* are needed if we do not want to allow these types of executions. A *liveness condition* L for a timed automaton A is a subset of the timed execution fragments of A such that any finite timed execution of A has an extension in L. Formally, $L \subseteq$ *t-exec(A)* such that for all $\alpha \in t\text{-}exec^*(A)$ there exists an $\alpha' \in$ *t-frag(A)*, such that $\alpha{\cdot}\alpha' \in L$.

For a live GTA, we want to ensure that the automaton behaves properly independently of the behavior of the environment. This property is know as *receptiveness* and a formal definition can be found in [31]. Intuitively, one can think of a game between the timed automaton and its environment where each has turns to make moves. The moves of the environment are input actions the GTA, while the moves of the timed automaton are internal and output actions. A timed automaton is receptive if and only if it has a winning strategy

[1] In [31] the model is called *live timed I/O automaton.*

against its environment. A strategy in the timed model is a pair of functions (g, f). Function f takes a finite timed execution and decides how the system behaves till its next locally-controlled action under the assumption that no inputs are received in the meantime; function g decides what state to reach whenever some input is received. A winning strategy does not collaborate with its environment to generate a Zeno timed execution. A strategy is called Zeno-tolerant if it guarantees that the system never chooses to block time in order to win its game against the environment. That is, a Zeno-tolerant strategy produces Zeno timed executions only when applied to a Zeno timed environment, but the system does not respond to Zeno inputs by behaving in a Zeno fashion. Denote by $t\text{-}exec^{Zt}(A)$ the set of Zeno-tolerant timed executions of A. The reader is referred to [31] for all the details.

We can now define a *live GTA*. It is a pair (A, L) where A is a GTA[2] and $L \subseteq t\text{-}exec^{\omega}(A)$, such that the pair $(A, L \cup t\text{-}exec^{Zt}(A))$ is receptive. This definition of the liveness property L is more general than is needed for our work. For the work in this chapter we only require and use the special case where the liveness property $L = t\text{-}exec^{\omega}(A)$. Therefore, for each live GTA we describe later in this work, the liveness condition is equal to the set of admissible timed executions of the GTA.

In Chapter 3 we defined what it means for two GTA's to be *compatible*, and we also defined the *parallel composition* operator, $\|$, for compatible GTA's. Two live GTA's (A_0, L_0) and (A_1, L_1) are compatible if A_0 and A_1 are compatible. We would also like to be able to perform parallel composition on compatible live GTA's, and the operation should be closed. That is, if (A_0, L_0) and (A_1, L_1) are compatible and $(A_0, L_0)\|(A_1, L_1) = (A, L)$, then (A, L) should also be a live GTA. In [31] the parallel composition operator is defined for live GTA's and the proof that the operator is closed is also presented there.

11.2.6 Live CGTA

To get the liveness property we want and local clocks in the model, we combine the CGTA with the liveness property from the live GTA model to get the *live CGTA* model. For the proofs later in this work, we need a liveness property that relates admissible timed executions of live GTA to clock functions. Informally speaking, we want that for every clock function

[2] In [31] A is called *safe timed I/O automaton.*

that can be applied with the fix-clock operator to a CGTA, if the environment is non-Zeno, then there exists a timed execution where the time of the local clock is not blocked. This requirement is captured formally in the following definition of live CGTA.

Definition 11.2 (Live CGTA)
A live CGTA is a pair (A, L) such that for every clock function cf, *(fix-clock(A, cf), L)* is a live GTA. □

We model the client and server as live CGTA and the channels as live GTA. When we describe a particular execution of the system, we apply clock functions to the CGTA to get values for *clock* variables. The parallel composition of the client, channels, and server forms the system. Because parallel composition of live GTA is closed [31], the resulting composed system is also a live GTA.

11.2.7 The projection operation

Before we describe the different components of the system, we define the projection operation on timed execution fragments that are from timed automata that are formed by the composition of timed automata. Let α be a timed execution fragment of a timed automaton that is the composition of timed automata, and let A be one of the component timed automata. Define the *projection* of α on A to be the sequence obtained by projecting all states of the composed system onto those of A and removing actions not belonging to A. We use the notation, $\alpha|A$, for the result of this operation. Informally, $\alpha|A$ is automaton A's view of timed execution fragment α. If $\alpha|A$ differs from $\alpha'|A$ only because of the splitting and combing of time-passage actions, then these are essentially the same views. Recall that in Chapter 3 we defined what it means for two timed execution fragments to be *time-passage equivalent*. Later in the chapter we refer to the time passage equivalence of projections of timed execution fragments.

11.2.8 μ-SLL-FIFO channels

The communication channels have the following properties.

1. Packets[3] placed in a channel are delivered in FIFO order.

2. Packets are not duplicated.

3. There is a maximum packet lifetime μ which is an upper bound on how long a packet can stay on a channel before it is either received or dropped.

4. If infinitely many copies of a packet p get sent on a channel, then infinitely many copies of p are received. This property is the strong loss limitation (SLL) property of channels given by Lynch in [21].

11.2.9 The client and server hosts

The client and server hosts are modeled by live CGTA (C, L) and (S, L') respectively, where L is the set of admissible timed executions of C and L' is the set of admissible timed executions of S. The CGTA's C and S have the following properties.

1. Each has an infinite and stable set of UID's on which it can perform only the following operations:

   (a) *generate()* which nondeterministically returns a new UID from the hosts' set of UID's and removes that id from the set, so it cannot be used again, and

   (b) *same(x,y)* which returns true iff $x = y$, where x and y are UID's.

2. In an admissible timed execution where clock values are determined by clock functions, after a crash there is an eventual recovery that returns the crashed host to an initial state. We assume that the local clock is not affect by the crash.

Since we are concerned only with the delivery of messages from the client to the server, and for our proofs we need to allow crashes only at the server, we use the following user interface actions.

- *send(m)* is the input action at the client to send a message m.

- *deliver(m)* is the output action at the server that delivers m.

[3] We use the term "packet" to denote objects sent over the channels by a protocol, and the terms "message" or "data" for user-meaningful data.

- *crash* is the input action that signals a crash at the server.
- *recover* is the output action that indicates the server has recovered from a crash.

Additionally, both client and server can place packets on and receive packets from the channels. For the rest of this chapter of the thesis, when we use the term *client* and *server* we mean specifically the model described in this section.

11.3 The problem

We now present a formal definition of the problem that T/TCP was designed to solve. We call it the *at-most-once fast delivery* problem.

Definition 11.3 (The at-most-once fast delivery problem)

at-most-once, in order delivery Messages from the client user are always delivered at most once and in the right order. That is, for every execution there exists a function *cause* that maps *deliver* actions to preceding *send* actions such that:

1. For every *deliver* action π, π and $cause(\pi)$ have the same message argument.
2. *cause* is one-to-one (at-most-once property).
3. For any two *deliver* events π_1 and π_2, if π_1 precedes π_2, then $cause(\pi_1)$ precedes $cause(\pi_2)$ (in order property).

eventual delivery In an admissible timed execution where clock values are determined by clock functions the following conditions hold:

1. If there are no crashes then all messages are delivered.
2. If there are finitely many crashes, messages sent after the last *crash* action and the subsequent *recover* action are eventually delivered.

fast delivery For any admissible timed execution in which there is a *deliver(m')* (for any message m') action at the server and the client subsequently receives a *send(m)* input, if the following conditions hold:

1. The clocks of the client and server always (through the whole execution) run at the rate of real time. That is, the clock function for both the client and the server is the identity function.

2. Both sides are recovered at the time of the *deliver(m′)* action, and there is no *crash* or *recover* event on either side after it.

3. Any packet sent by the client or server after the client receives the *send(m)* action from the user takes time at most d to arrive at its destination.

Then the server performs *deliver(m)* in time strictly less than $2d$ after the client receives the *send(m)* input. □

Notice that we weaken the delivery requirement from the one trip across the network (d) required for efficient transactions, to strictly less than $2d$. We can weaken the problem statement in this manner because the key property we need in our proof is that there are executions where the client does not receive any packets from the server, after it gets the *send(m)* input, before it finishes sending the packets that cause the message to be delivered within time $2d$. We are not requiring quiescence on the part of the server as is the case in the models presented in [17]. However, there are executions in the model where all the packets sent by the server after the last *deliver* event and before the *send(m)* input get dropped from the channel, and any packets sent by the server after the *send(m)* input takes time d to arrive at the client. In such an execution, even though the server does not quiesce, the client does not receive any packets from the server after it gets the *send(m)* input and before it finishes sending the packets that cause the message to be delivered within time $2d$.

11.4 Impossibility of at-most-once fast delivery

We can now state and prove the impossibility result.

Theorem 11.1

No system consisting of μ-SLL-FIFO channels and client/server hosts can solve the at-most-once fast delivery problem.

Proof: In our proof we use the general strategy employed by Kleinberg et al. in [17]. That is, we construct executions that behave as required by the problem definition, and then show that we can construct another execution that is a sort of combination of the previous executions, but where the new execution has incorrect behavior.

We start by assuming we have a protocol that solves the at-most-once fast delivery problem, and show that this assumption leads to a contradiction. Throughout the proof we mention the real time at which different events occur even though the client and server do not have access to real time. The local clocks are $clock_C$ and $clock_S$ for the client and server respectively. In an execution, the values for these clocks are determined by the clock functions we describe. In all the executions we construct $clock_C$ is equal to real time; that is, for all the executions we construct, the clock function of the client is the identity function.

The first execution we construct, α_1, is shown in Figure 11-1. In this execution $clock_S$ is also equal to real time, which means the clock function of the server is the identity function. The client receives a *send(m′)* input at real time 0. All packets sent by both the client and server take time d. The parallel composition of the client, the channels, and the server forms the system. Since each component is a live GTA, we know by the closure result of [31] that the composed system is also a live GTA. Therefore, since the environment in execution α_1 is non-Zeno, we know that α_1 is an admissible timed execution. The same argument holds for all the subsequent executions we construct in this proof.

Since both the client and server are recovered at time 0, α_1 is an admissible timed execution, and there are no *crash* or *recover* actions after the *send(m′)* input, the eventual delivery property results in the server action *deliver(m′)*. Let $clock_S = p$, which is also real time p, be the time of this action. For this execution the client and server uses the set of id's $usedids_c^1$ and $usedids_s^1$ respectively.

Now we construct a second execution α_2 shown in Figure 11-2. Again in this execution $clock_S$ is equal to real time. This execution starts out the same as execution α_1. That is, the client receives a *send(m′)* input at real time 0, and packets sent by the client and server starting at time 0 take time d to arrive. However, execution α_2 starts to differ from execution α_1 after time $p-2d$ on the server side and $p-d$ on the client side. The difference is that packets sent strictly after time $p-2d$ on the server side get dropped from the channel

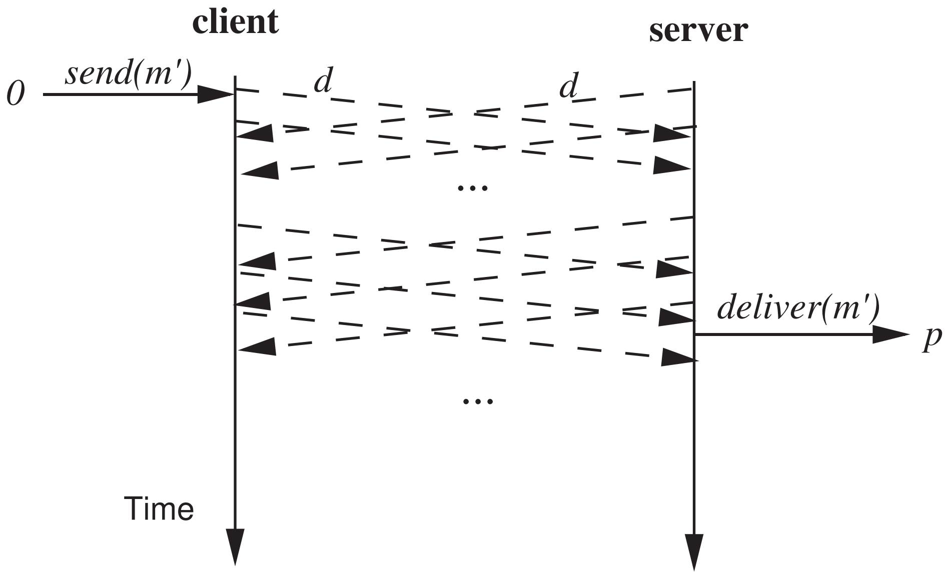

Figure 11-1: ***Execution*** *α_1. The numbers outside the time lines are real time, the dashed lines represent packets, and the "$\cdots$" between packets represents a finite number (whatever the protocol needs) of packets in both direction. The numbers on the dashed lines represents the time it takes the packets to traverse the channel.*

and packets sent strictly after $p - d$ on the client side also get dropped from the channel. In this execution the client uses the set $usedids_c^2$ of UID's and the server uses the set $usedids_s^2$, where $usedids_c^2$ and $usedids_c^1$ are disjoint and $usedids_s^2$ and $usedids_s^1$ are also disjoint. During the time interval from the *send(m')* input to the real time $p - d$, the client can receive the same number of packets from the server as it did in execution α_1 because packets that arrive at the client by time $p - d$ must be sent by the server no later than time $p - 2d$. Similarly, in the real time interval $[0, p]$ the server can receive the same number of packets in this execution as it did in execution α_1 because packets that arrive by time p must have been sent by the client by time $p - d$.

The packets sent between the client and server in this execution have different id's from the packets sent in execution α_1, but the id's can be used in the same way. Recall that client and server can only generate id's from their infinite set of UID's, and can only test if a received id is the same as some other id. Because of these restrictions on how id's can be used, a host cannot know before it receives any packets from the other host what id's the other host is going to use. Also, after a host receives a series of packets, the most information it can have about the id's that are going to be on subsequent packets is whether they should

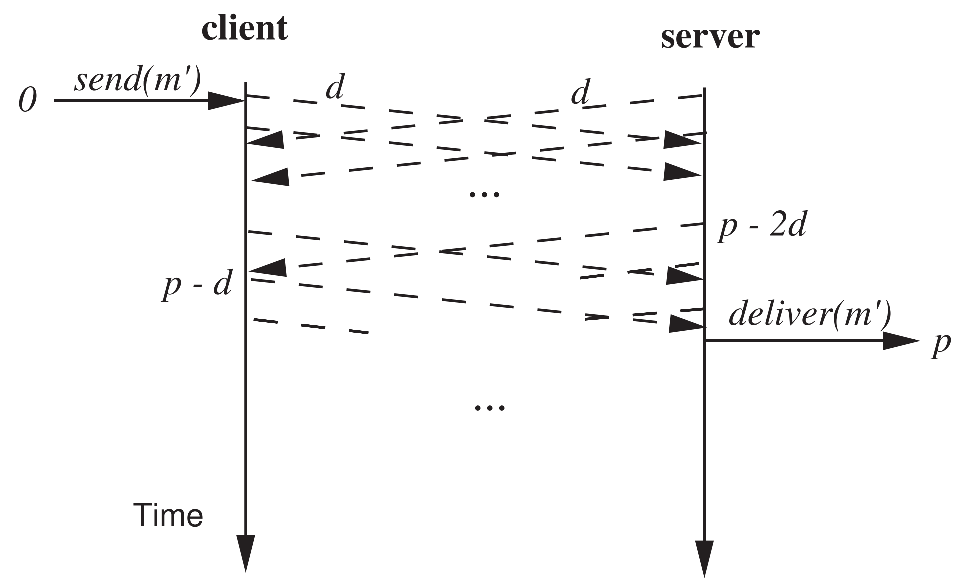

Figure 11-2: ***Execution*** *α_2. Same as execution α_1, except dashed lines that do not go all the way across represent packets dropped by the channels.*

be the same as and/or different from id's already received. Therefore, if in execution α_2 a host receives a packet with id u and performs the operation *same(u,v)* for some id v, and if at the same time in execution α_1, the same host receives a packet with id x and performs the operation *same(x,y)* for some id y, then *same(u,v)* = *same(x,y)*. Thus, the fact that $usedids_c^2$ and $usedids_s^2$ are used in execution α_2 does not affect the behavior of α_2 relative to α_1. Therefore, in execution α_2, at time p the server can perform the *deliver(m′)* action.

Throughout the rest of this proof we compare executions where packets are sent and received at the same local clock time, but where the packets have different sets of UID's. The argument just presented can be applied to all these comparisons to show that the use of a different set of id's cannot cause the client or server to behave differently.

The next execution we construct is α_3. It is shown in Figure 11-3. Again $clock_S$ is equal to real time. This execution starts out the same as execution α_2. That is, the client gets a $send(m')$ input at real time 0 and sends packets that take time d to arrive with packets sent after time $p-d$ being dropped from the channel. On the server side the packets sent up to time $p-2d$ also take time d to arrive and packets sent after are dropped. For this execution the client uses the set $usedids_s^3$ of UID's and the server uses the set $usedids_s^3$. Apart from the difference in id's used, for the real time interval $[0,p]$ $\alpha_3|C$ time passage equivalent to

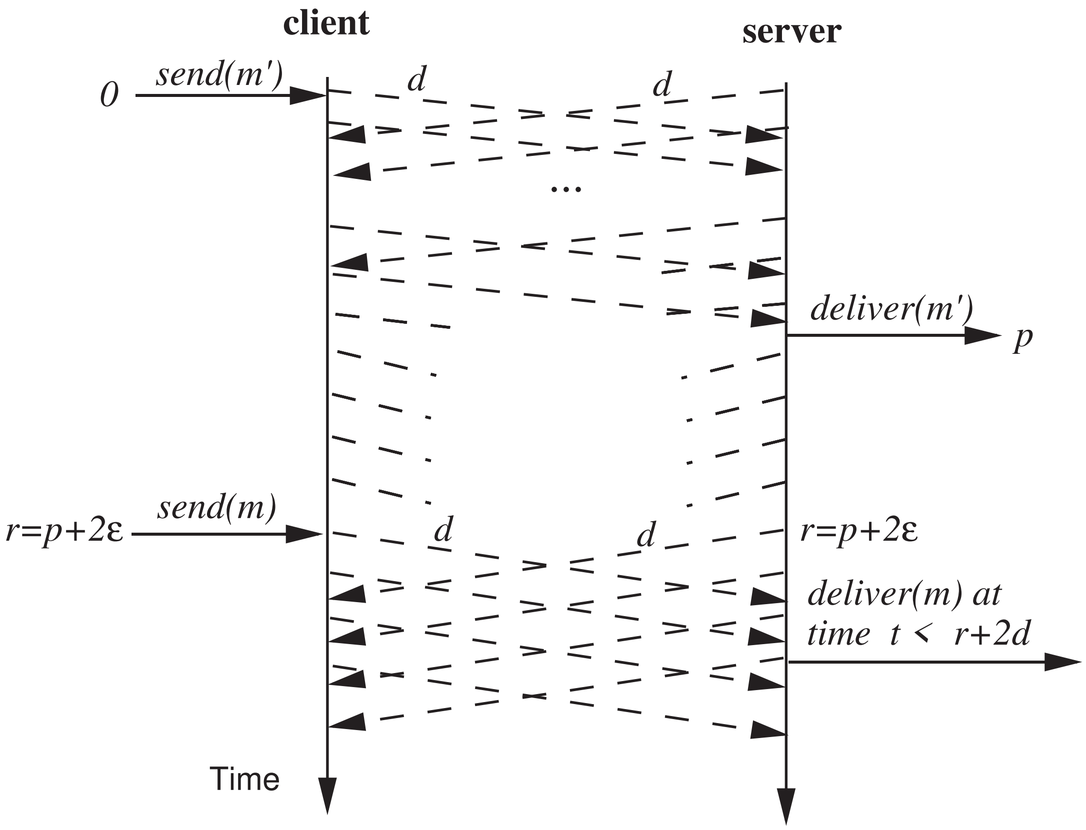

Figure 11-3: ***Execution*** α_3*. This execution is an extension of* α_2 *that includes a second deliver action.*

$\alpha_2|C$, and for the real time interval $[0, p]$ $\alpha_3|S$ time passage equivalent to $\alpha_2|S$. Therefore, at time p the server can perform the $deliver(m')$ action.

Execution α_3 continues as follows. At real time $p + 2\epsilon$, where ϵ is an arbitary constant greater than 0, the client receives a *send(m)* input and all packets sent by both the client and server after this input action take time d to arrive. Because the value $p + 2\epsilon$ appears in several subsequent executions we construct, to simplify the notation we let $r = p + 2\epsilon$. Since we assume the protocol satisfies the fast delivery property, at some real time $t < r + 2d$ the server delivers the message m.

Now consider the execution α_4, shown in Figure 11-4. This execution is exactly the same as execution α_3 except that the client and server use the set of UID's $usedids_c^4$ and $usedids_s^4$ respectively, all the packets sent by the server after the *send(m)* event get dropped by the channel, and all packets sent by the client at or after time $r + d$ also get dropped

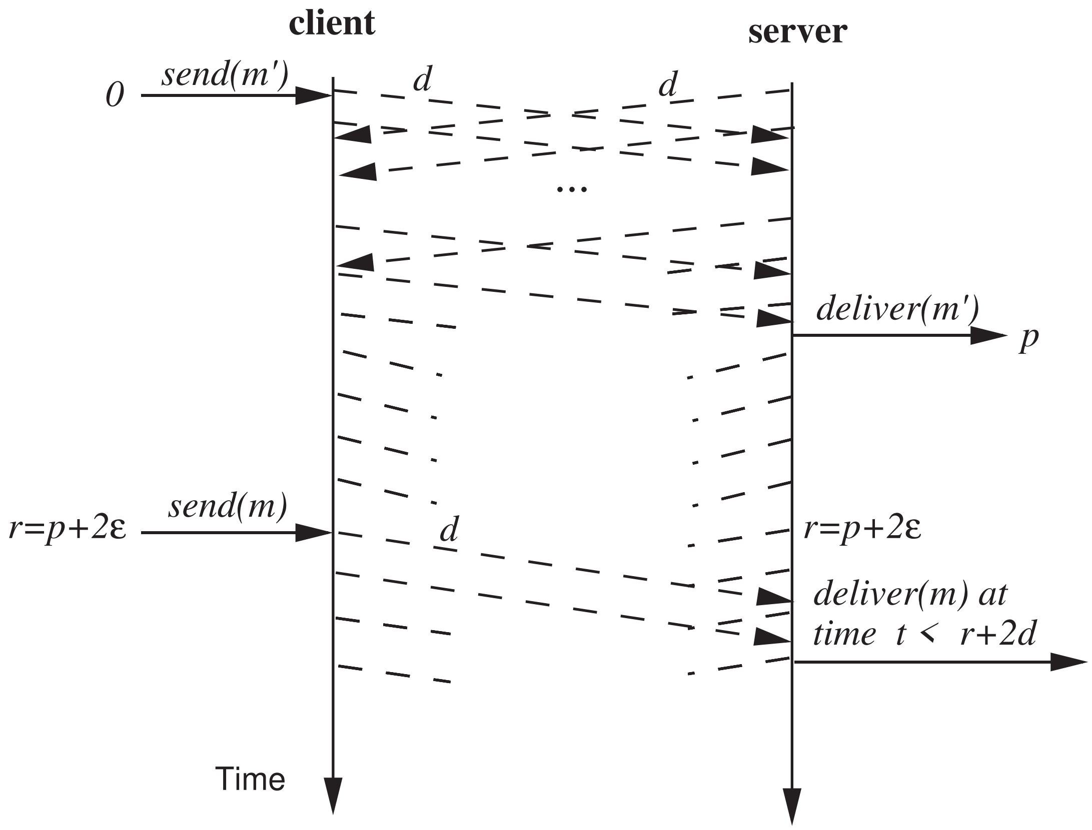

Figure 11-4: ***Execution*** *α_4. This execution is the same as α_3 except that additional packets are dropped from the channels.*

from the channel. However, from time 0 up to and including time t, modulo the id's, $\alpha_4|S$ time passage equivalent to $\alpha_3|S$. The executions look the same to the server for that time interval because it receives the same number of packets from the client and at the same time in both executions. Also for both executions the packets sent by the client cannot contain any information about packets received from the server that were sent at or after time r because all the packets that the client sends that arrive at the server by time t must have been sent before time $r+d$; therefore, in both execution α_3 and execution α_4, these packets are sent before any packet sent by the server after the *deliver(m')* event is received by the client. Thus, at time t in execution α_4 the server can deliver m. The bound of less than $2d$ on delivery time is important here because it forces the server to deliver the message even though the client has not have received any packets from it since the *send(m)* event.

The next execution α_5 is shown in Figure 11-5. For parts of this execution $clock_S$ runs at the rate of real time and for other parts it runs faster than the rate of real time. We define the clock function for the server by giving the rate of $clock_S$ relative to real time for different real time intervals. For the real time interval $[0, p]$, $clock_S$ runs at the rate of real time. In execution α_5 the client uses the set of UID's $usedids_c^5$ and the server uses the set $usedids_s^5$. Even though in this execution the id's used by the client and server after the *send(m′)* are different from the ones used in execution α_2, again at time p the server can perform the *deliver(m′)* action.

After the *deliver(m′)* action and up to real time $p + \epsilon$, that is, for the interval $(p, p + \epsilon]$, $clock_S$ runs at $(2\epsilon + 2d)/\epsilon$ times the rate of real time, and from time $p + \epsilon$ through the rest of the execution, that is, the interval $(p + \epsilon, \infty)$, $clock_S$ runs at the rate of real time again. Now let the server receive a *crash* input at real time $p + \epsilon$. Because of the rate of $clock_S$ for the interval $(p, p + \epsilon]$, at real time $p + \epsilon$, $clock_S = p + 2\epsilon + 2d = r + 2d$. Since α_5 is an admissible timed execution and $clock_C$ and $clock_S$ are determined by clock functions, the server eventually recovers. The time of recovery is determined by the protocol, but it must happen after the *crash* event. Let k be the $clock_S$ time between crash and recovery. Since $clock_S$ is now running at the rate of real time, k is also the difference in real time between the *crash* input and the *recover* output. Thus, the recovery happens when $clock_S = r + 2d + k$, which is real time $p + \epsilon + k$.

The next execution, α_6, shown in Figure 11-6, starts out like execution α_5 except the client and server use the sets of UID's $usedids_c^6$ and $usedids_s^6$ respectively, and the clock function of the server is different from the clock function in execution α_5. The clock functions are the same for the real time interval $[0, p+\epsilon)$; that is, for the real time interval $[0, p]$ $clock_S$ runs at the rate of real time, and for the real time interval $(p, p+\epsilon]$, $clock_S$ runs at $(2\epsilon+2d)/\epsilon$ times the rate of real time. However, for the real time interval $(p + \epsilon, p + 2\epsilon]$, $clock_S$ runs at k/ϵ times the rate of real time. Therefore, in execution α_6 when $clock_S = r + 2d + k$ it is real time r. Because of the real time independence property, we know that when $clock_S$ $= r + 2d + k$ in this execution, the server can perform the *recover* action.

After the *recover* action through real time $r + 2d + k$, that is, the real time interval $(r, r + 2d + k]$, $clock_S$ runs at $d/(2d + k)$ times the rate of real time. Therefore, at real time

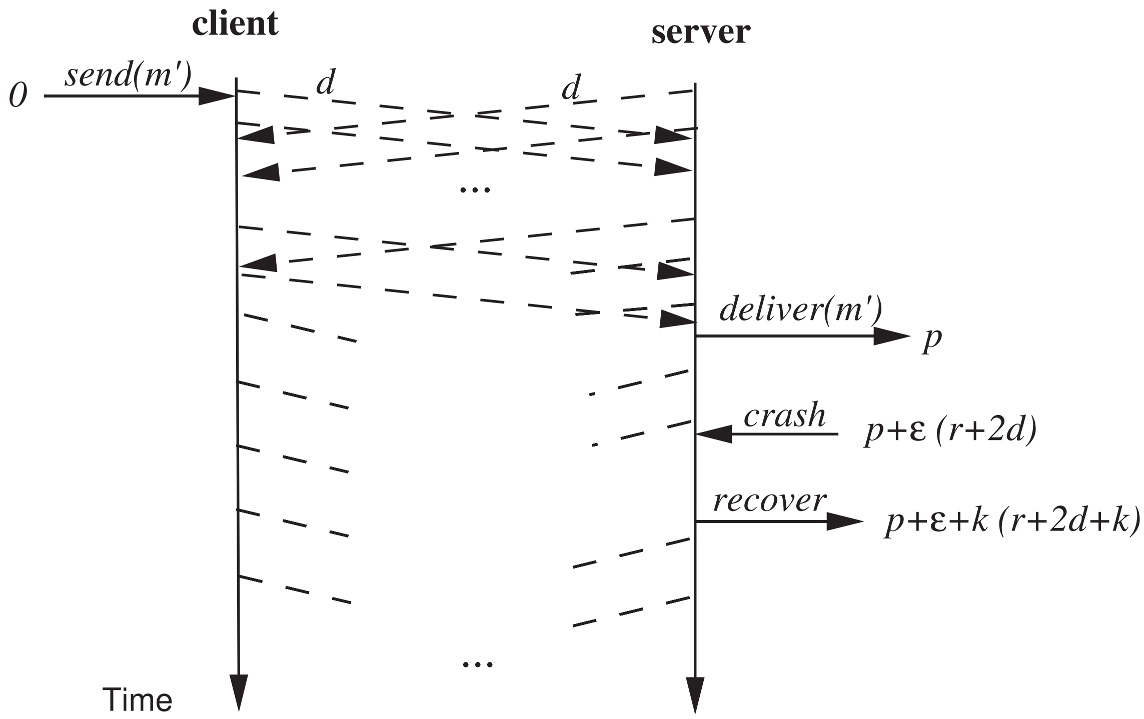

Figure 11-5: ***Execution*** α_5. *Values of* clock_S *are shown in parenthesis. Dashed lines are packets, and dashed lines that do not go all the way across are dropped packets. Vertical ellipses represent a finite series of packets.*

$r + 2d + k$, $clock_S = r + 3d + k$. After that time through the rest of the execution, that is, the real time interval $(r + 2d + k, \infty)$, $clock_S$ runs at the rate of real time. All of the packets that both the client and server send from real time r up to, but not including real time $r + 2d + k$ are dropped from the channels. However, starting at real time $r + 2d + k$ all packets sent by the client and server do not get dropped from the channels and take time 0 and d to arrive respectively. Note that when the server receives the packet form the client at $clock_S = r + 3d + k$ it cannot perform the *deliver(m)* for any m at or after this time. It cannot perform this action because *send(m′)* is the only *send* action in α_6, so any *cause* function has to map both *deliver(m)* and *deliver(m′)* to *send(m′)* which violates the at-most-once property which we assume the protocol satisfies.

The next execution α_7 is shown in Figure 11-7. This execution starts out like α_6, but the client and server use the sets $usedids_c^7$ and $usedids_s^7$ of UID's respectively. The server's clock function for this execution is the same as for execution α_6. That is, it runs at the rate

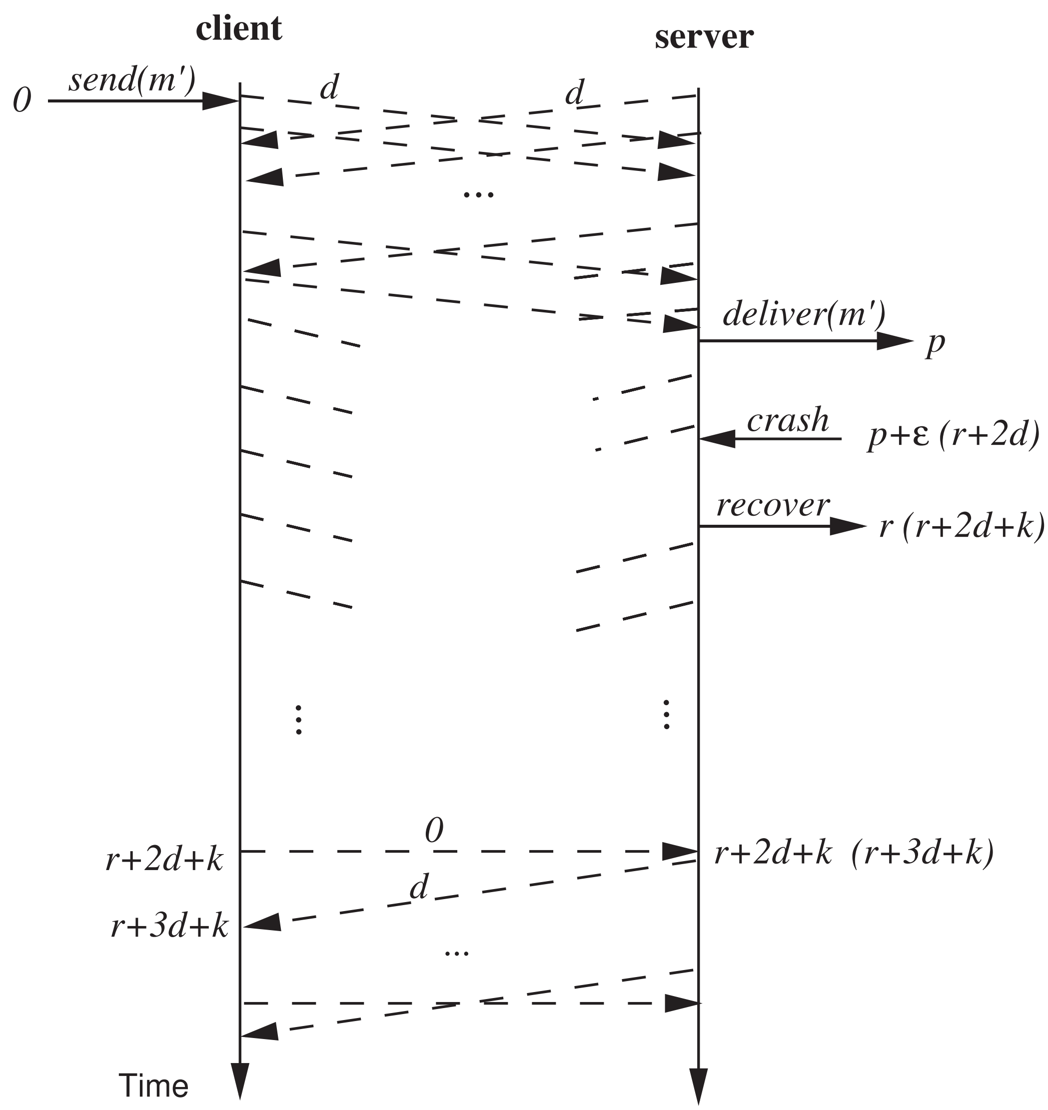

Figure 11-6: ***Execution*** α_6. *This execution is an extension of* α_5 *to include some additional sending and receiving of packets, and the clock function of the server is also different.*

of real time for the interval $[0, p]$, at $(2\epsilon + 2d)/\epsilon$ times the rate of real time for the real time interval $(p, p + \epsilon]$, at k/ϵ times the rate of real time for the real time interval $(p + \epsilon, p + 2\epsilon]$, at $d/(2d + k)$ times the rate of real time for the real time interval $(r, r + 2d + k]$, and at the rate of real time for the interval $(r + 2d + k, \infty)$. Here again the server can perform the *recover* action when $clock_S = r + 2d + k$.

After the *recover* action is where α_7 begins to differ from α_6. After the *recover* event at the server, the client gets the *send(m)* input at real time r at which time $clock_S = r + 2d + k$.

However, as in α_6, in α_7 all of the packets that both the client and server send from real time r up to, but not including real time $r+2d+k$ ($clock_S = r+3d+k$) are dropped from the channels. For the $clock_S$ interval $[0, r+3d+k]$, $\alpha_6|S$ time passage equivalent to $\alpha_7|S$. Thus, up to $clock_S = r+3d+k$ in α_7 the server cannot deliver m, because m cannot be delivered in α_6.

Again, as in α_6, all packets sent by the client and server starting at real time $r+2d+k$ do not get dropped from the channels and take time 0 and d to arrive respectively. Execution α_7 is an admissible timed execution, $clock_C$ and $clock_S$ are determined by clock functions, and m is sent after the last *crash* and *recover* actions. Therefore, since the protocol satisfies the *eventual delivery* property the server must eventually perform the *deliver(m)* action. Let real time t' and $clock_S = t'+d$ be the time of this event.

Finally, we construct an execution α_8 where the server delivers the same message twice. This execution is shown in Figure 11-8. In the execution $clock_S$ runs at the rate of real time for the whole execution, that is, the clock function of the server is also the identity function. In execution α_8 the client uses the set of UID's $usedids_c^8$ and the server uses the set $usedids_s^8$. On the client side, except for the use of a different set of id's, execution α_8 is exactly the same as execution α_7, so *send(m′)* happens at time 0, and *send(m)* happens at real time r. However, in execution α_8 the packets the client sends after the *send(m)* input and before time $r+d$ are not dropped from the channel. On the server side, except for the use of a different set of id's, from time 0 to time t execution α_8 looks the same as execution α_4. That is, modulo the id's, for the time interval $[0,\ t]$ execution $\alpha_8|S$ time passage equivalent to $\alpha_4|S$. Therefore, since in execution α_4 the server performs *deliver(m′)* at time p and *deliver(m)* at time t, in execution α_8 it can do likewise.

For the rest of α_8, the packets the client sends at or after time $r+d$ until, but not including time $r+2d+k$ are dropped from the channel, and on the server side at time $r+2d$ a *crash* input occurs. For the real time interval $[r+2d, r+2d+k]$, $\alpha_8|S$ time passage equivalent to $\alpha_7|S$. Therefore, because of the real time independence property, at $clock_S = r+2d+k$, the server can perform the *recover* output action. Any packet sent by the server after the *recover* event up to, but not including time $r+3d+k$ is dropped from the channel. The packets that the client sends starting at time $r+2d+k$ take time

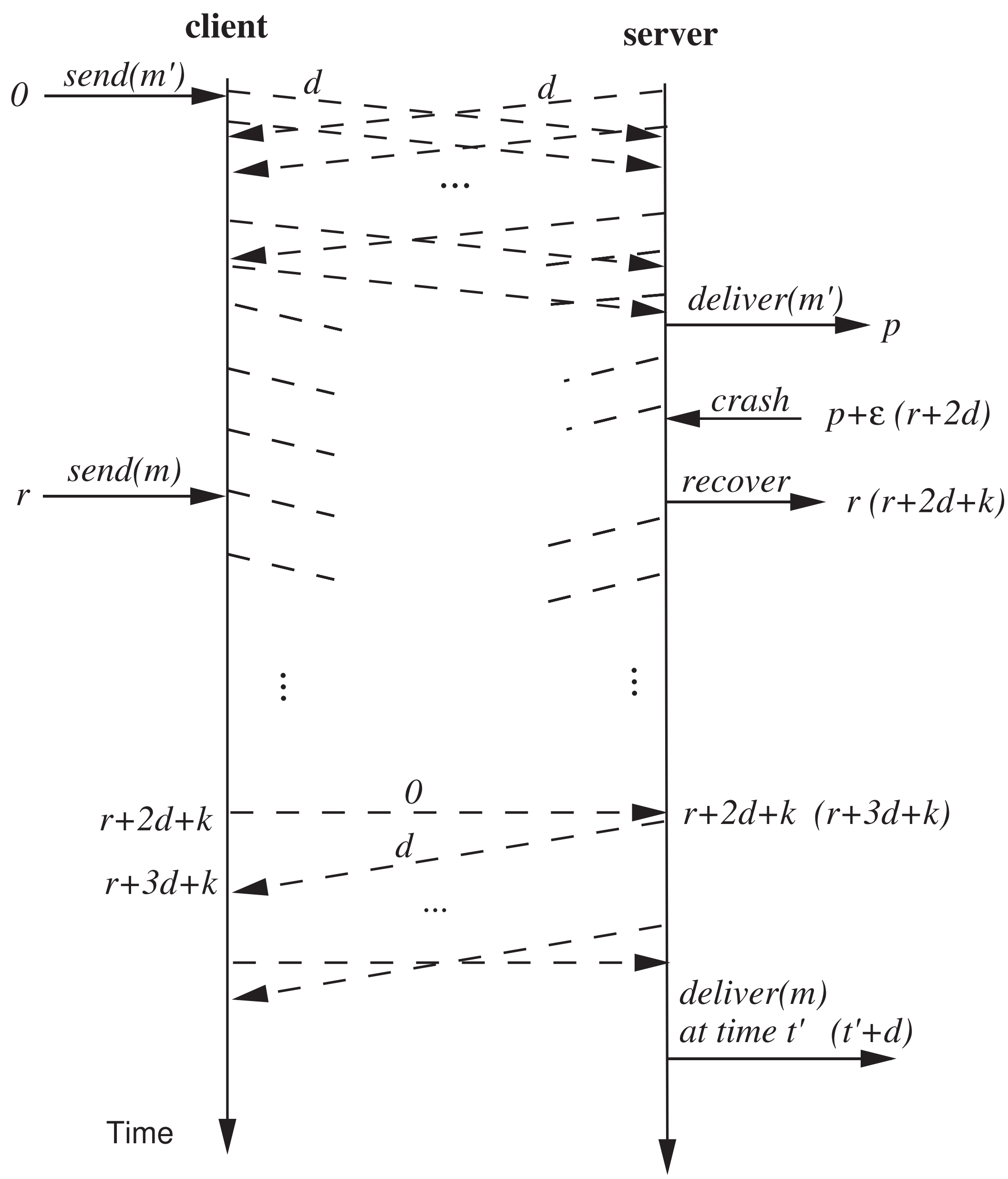

Figure 11-7: ***Execution*** *α_7. This execution is the same as execution α_6 except it includes a send(m) action and the subsequent delivery of m.*

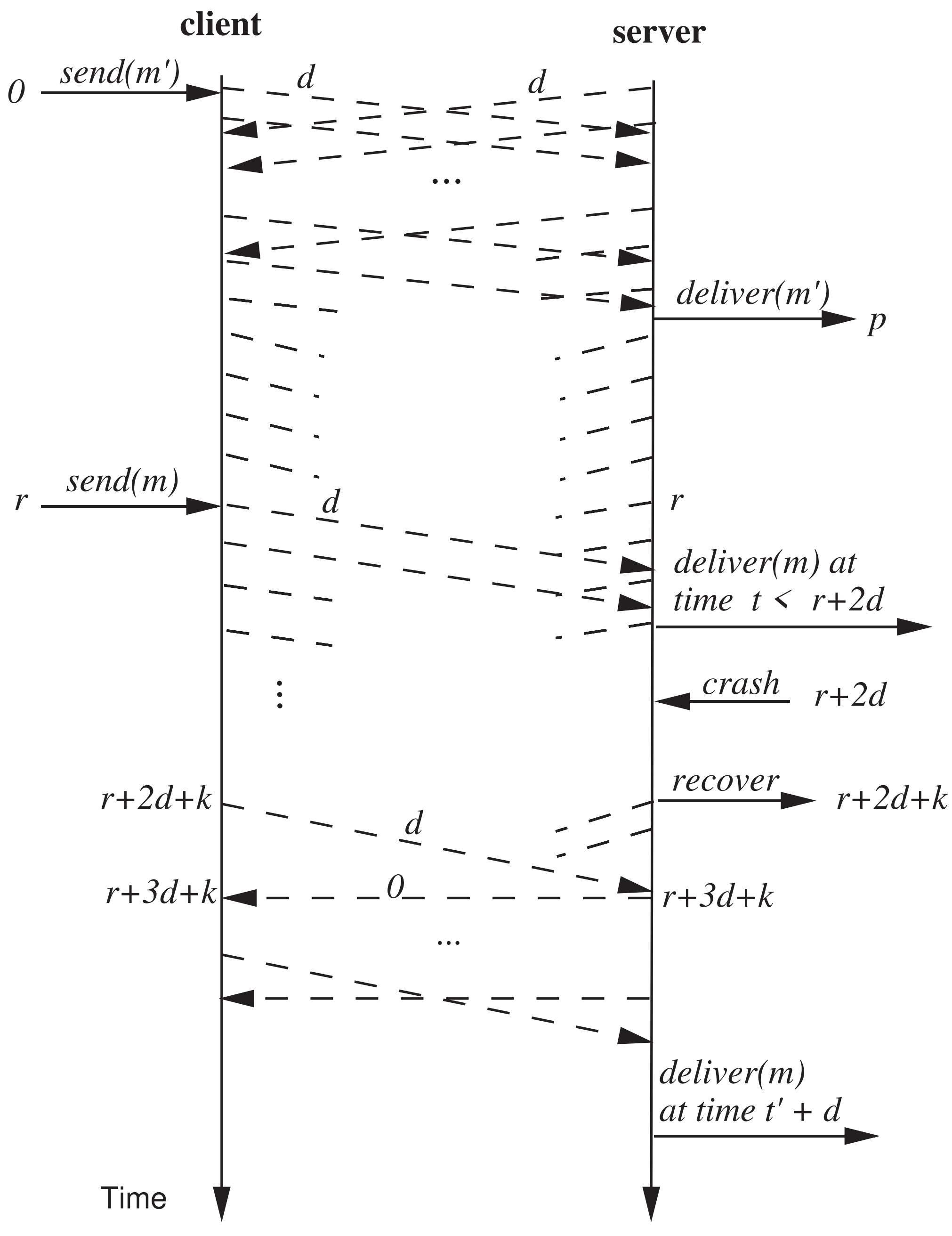

Figure 11-8: ***Execution*** α_8. *This execution demonstrates how the at-most-once delivery property can be violated.*

d to arrive at the server, and the packets that the server sends starting at time $r + 3d + k$ take time 0 to arrive at the client. Except for the fact that packets sent and received have different id's, in the $clock_S$ interval $[r+2d,\ t'+d]$ in execution α_8 the server receives exactly the same inputs as in the same $clock_S$ interval in execution α_7. In this interval it receives a *crash* input from the environment, and the inputs from the channel (packets from the client) are the same (except for the id's) because modulo packet id's, $\alpha_8|C$ time passage equivalent to $\alpha_7|C$, and the only packets from the client that reach the server in the $clock_S$ interval $[r + 2d,\ t' + d]$ in both executions, are the ones that the client starts sending from time $r+2d+k$. These packets start arriving when $clock_S = r+3d+k$ in both executions. Since the *recover* action returns the server to an initial state where it does not remember any previous actions in both executions, modulo packet id's, $\alpha_8|S$ time passage equivalent to $\alpha_7|S$ for the $clock_S$ intervals $[r+2d,\ t'+d]$. Because of the real time independence property of the server, we know that at $clock_S = t'+d$ the server can perform the *deliver(m)* action. Since m was already delivered, we have duplicate delivery which contradicts our assumption that the protocol solves the at-most-once fast delivery problem. ■

11.5 Discussion of proof

In this section we provide some intuition for the proof by explaining the reasons for the times we choose to have events occur in executions α_7 and α_8. In execution α_7 we want the protocol to deliver m, so the *send(m)* event must occur at or after the *recover* event. In α_8 the *send(m)* event should happen at the same real time as in α_7. However, now we want m to be delivered as it was in execution α_4. This delivery takes time strictly less than $2d$ after the *send(m)* event. The delivery must be followed by a crash and recovery. The *recover* event happens at least $clock_S = k$ time after the delivery of the message. The crash and recovery in α_8 must happen at the same $clock_S$ time as in α_7. Therefore, since in α_8 the clock function of the server is the identity function, the *recover* event in α_7 and the *recover* event in α_8 are at least $2d+k$ apart in real time. Therefore, in α_7 at the time of the *recover* event, $clock_S$ must be ahead of real time by at least $2d+k$.

In α_7 messages by the client after time r and up to time $r+2d+k$ get dropped because this is the time frame in α_8 where the server delivers the message, crashes, and recovers. Packets sent at or after real time $r + 2d + k$ do not get dropped in α_7 because the server recovers at this time in α_8, so packets received by the server in both executions at or after this time are the first packets the server receives after recovering. In order for the proof to work, the packets sent by the client at or after real time $r + 2d + k$ must arrive at the server at the same $clock_S$ time in both executions α_7 and α_8. Similarly, the packets sent by the server at or after real time $r+2d+k$ must arrive at the client at the same $clock_C$ time in both executions. Since $clock_C$ remains unchanged in both executions while $clock_S$ does change, the delivery time of the packets sent at or after real time $r + 2d + k$ in α_8 must be different from the delivery time of packets sent at or after real time $r + 2d + k$ in α_7 in order for them to arrive at the same $clock_S$ and $clock_C$ times in both executions. In α_7 at or after real time $r + 2d + k$, $clock_S$ is ahead of real time by a fixed amount, x, and in α_8 clock$_S$ is equal to real time. Therefore, in α_8 packets sent by the client at or after real time $r + 2d + k$ must take the additional time of x to arrive at server. Similarly, packets sent by the server at or after real time $r+2d+k$ in α_7 must be sent at the same $clock_S$ time in α_8. Therefore, in α_8, they are sent later in real time by x, which means for them to arrive at the client at the same time in both executions, their time on the channel in α_8 must be x less that their time on the channel in α_7.

In our proof we let $x = d$. Thus, in order to allow d to be any value from 0 to μ, we set the delivery time for packets from the client and the server at or after real time $r + 2d + k$ in α_7 to be 0 and d respectively. If we did not want to allow 0 time delivery of packets in the model, we can let $x < d$. For example, if we let $x = 0.5d$, then we can let the delivery time for packets from the client and the server at or after real time $r + 2d + k$ in α_7 to be $0.5d$ and d respectively. Then in α_8 the delivery times would be d and $0.5d$ respectively.

While it is impossible for any protocol to solve the at-most-once fast delivery problem, in practice an even weaker correctness condition may be sufficient. In the next chapter we present a weaker specification for the problem T/TCP is supposed to solve, and shows that T/TCP implements this specification.

Chapter 12

Conclusion

12.1 Summary

In this thesis we presented three very large case studies of the use of *simulation* and *invariant assertion* techniques for the formal verification of complex distributed protocols. We verified two versions of TCP and a version of the experimental transport level protocol called T/TCP. T/TCP is designed to provide the reliable data streaming of TCP, while being more efficient for transactions. However, under certain circumstances T/TCP may deliver the same message twice. We showed in the thesis that it is in fact impossible for any protocol to provide reliable data service and fast transactions if the client and server have clocks that may run at arbitrary rates.

12.1.1 Verification of protocols

TCP is designed to provide reliable transport level service. The first step in verification is to write a precise formal specification for this problem. We write the specification using the untimed automaton model [24]. The key idea in the abstract specification is to represent data sent during an incarnation of a connection as elements of a FIFO queue. Each of the client and server has an infinite array of these queues. The array of queues at the client and server are indexed by the infinite set of unique identifiers (uid's) at the client and server respectively. For each new incarnation the client and server both chose a new id. The client uses its id as the index for the queue on which it sends data for that particular incarnation, and the server does likewise. To form a connection, id's chosen by the client and server are non-deterministically paired together. Each id can only be paired with one other id from the other host. A host can only receive data from a queue if its current id is paired with the id of the sender of the data, and since each id can only be associated with one other id, a host can only receive data from a unique incarnation during the life of that incarnation.

After presenting the specification for the reliable transport level problem, we presented a formal model for TCP, but with the assumption that it has unbounded and stable counters. The protocol is described using the General Timed Automaton model [21]. We then use simulation techniques [24, 26] to formally verify that TCP with unbounded counters implements the specification for the reliable transport level problem. However, in reality TCP does not have unbounded and stable counters. It instead uses a 32 bit clock based counter that cycles in approximately 4.5 hours for initial sequence number generation. It also uses a 32 bit number cyclic space for numbering each byte of data sent. Therefore, in practice TCP relies on timing properties of the counters, the maximum segment lifetime, and a series of timeouts to give the illusion of an unbounded set of sequence numbers. In the thesis we clarify the timing properties that are needed, and show executions where incorrect behavior results if the correct timeouts are not used. The official TCP references [28, 30] are somewhat unclear on which timeouts are necessary for correct behavior. Both references describe timeouts that are necessary for correct behavior as being optional. The duration of quiet time after crashes specified in [28] is also not long enough and can lead to incorrect behavior in some situations.

We described TCP with bounded counters and the correct timeouts using the GTA model, and then used a forward simulation to show that it implements a slightly modified version of TCP with unbounded counters. The modification allows non-deterministic timeouts in TCP, and we show the modified version of TCP still implements the specification. Thus, TCP with bounded counters and timeouts implement the specification for the reliable transport level problem.

After defining a specification for the problem and formally verifying both versions of TCP we next sought to show that T/TCP implements TCP. However, we observed that even with unbounded and stable counters, T/TCP does not satisfy the at-most-once semantics required of reliable transport level protocols. This fact had been shown earlier by Shankar and Lee in [33]. The designers of T/TCP and other network protocol designers think that in some situations it is acceptable for the same data to be delivered twice, so the behavior of T/TCP is not necessarily wrong. Therefore, in the thesis we present a weaker specification for the transport level problem. This weaker specification captures the different behaviors of T/TCP. The key differences in the weaker specification is that we allow the hosts to resume an incarnation after a crash or an abort, and we allow some data to be delivered more than once. In the stronger specification after a crash or abort new incarnations must be started. After presenting the weaker specification we formally verify that T/TCP implements this specification. The verification of this protocol follows the same pattern as the verification of TCP with unbounded counters.

12.1.2 Impossibility result

T/TCP does not solve the reliable transport level problem because the optimizations that make it efficient for transactions may also cause it to deliver duplicate messages after a crash. This observation caused us to wonder whether it is possible for any protocol to perform transactions efficiently and still have reliable data streaming. In the thesis we prove that under certain specific circumstances it is impossible for any protocol to do both. The formal automaton models we used for the verifications in the thesis are not sufficient for presenting the impossibility result. For the impossibility result we had to deal with the issue of liveness, since we do not want to allow protocols that solve the problem by blocking time. We also had to deal with issue of the client and server having local clocks, but no access to real time. The formal model used to describe the system and present the proof of impossibility is a novel combination of the live GTA model of Segala et al. [31], which can handle liveness issues, and the clock GTA model of De Prisco [29], which allows local clocks. We prove that in a system where the client and the server have local clocks and infinite and stable sets of unique identifiers, but not counters, that even if we require fast delivery only when the clocks are accurate and not immediately after crashes, it is impossible to have fast delivery and still satisfy the the at-most-once delivery property if the local clocks sometimes run at arbitrary rates.

12.2 Evaluation

The thesis has clearly demonstrated that simulation and invariant assertion techniques can be used to verify the correctness of complex real world protocols. We believe our abstract modeling of TCP and T/TCP captures the most significant aspects of the protocols related to safety. However, we do make some simplifying assumptions about the protocols. Most of the simplifications we make are related to the performance of the protocols. In our abstract model, every segment contains at most one byte of data. We also do not include the sliding window mechanism, so each byte of data needs to be acknowledged before new data is sent. The fact that in the real protocols segments can contain more than one byte of data, and that the sliding window mechanism allows multiple segments to receive a single acknowledgment are performance issues that we do not believe are critical to the safety of the protocols. Thus, even though we do not verify versions of TCP and T/TCP that include all the complexities, we believe our models for these protocols capture the essential properties related to safety.

Even though we do not include all the details of the protocols, the proofs are still very long. In particular, the verification of each of the three protocols require the proving of a large number of invariants. We often found that in order to prove one key invariant, several other auxiliary invariants that essentially "lead up" to the invariant we want to prove had to be proved before. For example, if we wanted to prove that when a segment with a certain sequence number is on a channel certain conditions are true, we often had to prove an some invariants about conditions in the states from which that segment could be sent. Many times we had to prove a sequence of three or four invariants before we could prove one key invariant. We found that a lot of the difficulty in the invariant proofs had to do with finding the right auxiliary invariants. Once the correct sequence of auxiliary invariants was found, the actual proof of each piece tended not to be too difficult.

However, even though each little piece of the proof may not be too difficult, the sheer size of the proofs may make protocol designers reluctant to use the formal methods used in this thesis. Therefore, the continued development of tools that automate the process of proving simulation relations and especially the proof of invariants is important for making the methods more practical. Formal verification methods are most interesting to protocol designers when they reveal flaws in protocols. In doing the work for this thesis we found that the formal methods we use can reveal flaws in the protocol even before all the small details are worked out. For example, our observation that T/TCP may deliver the same data twice occurred while we were trying to simulate the accelerated opening of T/TCP by a sequence of TCP steps. The observation came before we started to proving low level invariants of that protocol. Therefore, the formal methods used in the thesis can be useful to protocol designers even before all the details of the proofs are worked out.

12.3 Future Work

In the short term, we would like redo at least parts of the proofs in this thesis using an existing automated tool such as the Larch Prover [11]. We are particularly interested in using automated tools for the proofs of low level invariants. The precise structure of these proofs make them particularly amenable to automated verification. There are already several examples where the Larch Prover has been used for these types of proofs [34, 20]. Another short term project is to show that the impossibility result holds even when the client and server are allowed to use unbounded counters. Since T/TCP has duplicate delivery even though it uses unbounded counters, we believe counters do not help for any protocol. However, the proof we use in the thesis does not hold in the case where the process may have counters. The reason the proof as we have it now does not work for counters is that when the different executions are combined, the hosts many be able to distinguish the combined executions from the other excutions if they can do more sophisticated comparisons of the id's in addition to testing whether two id's are the same. With counters, the host are able compare whether one id is greater than, or less than other id's.

In the long term we would like to work on developing more tools for automating simulation and invariant assertional proofs. We are also interested in using the formal methods to verify additional network protocols. We not only want to use the methods to verify safety properties, but also want to use the formal methods to verify performance properties of protocols. The issue of quality of service guarantees is almost as important as safety for transport level protocols, and we believe the formal methods used in the thesis can be enhanced to analyze performance characteristics of protocols. In our work we find that the formal methods provide valuable insights into how protocols work and what aspects of the protocols are essential for correctness, so we believe these methods can be useful in the designing of protocols. Therefore, in the future we want to work on the design of protocols, where formal verification methods are incorporated in the design process.

Appendix A

A.1 Sets

We use standard notation for sets. A set consisting of the elements $e_1, e_2, \ldots$ we write as $\{e_1, e_2, \ldots\}$.

The empty set is denoted by $\emptyset$, set membership by $\in$, and $\subset$ and $\subseteq$ denote proper subset and subset relations respectively. We also use the standard set operators.

$\cup$ Union

$\cap$ Intersection

$^-$ Complement (with respect to some given set)

$\setminus$ Set minus.

We also use the notation $:\in$ to assign an arbitrary element of a set to a variable. That is, $v :\in \{e_1, e_2, \ldots\}$ means $v = e_1$ or $v = e_2, \ldots$.

A.1.1 Cardinality

The cardinality of a set S, is written $|S|$ and is defined as

$$|S| \triangleq \begin{cases} n & \text{if } S \text{ has } n \text{ elements} \\ \infty & \text{if } S \text{ has infinitely many elements} \end{cases}$$

A.2 Bags (Multisets)

For bags we use the following operations from the previous section:

$|s|, \cup, \cap, \in, \setminus$

$|s|$ counts the total number of elements including duplicates of s.

A.2.1 Bag Type

For any set S, denote by $\mathcal{B}(S)$ the set of all (finite or infinite) bags with elements from S.

A.3 Queues

A queue q consisting of the elements $e_1, e_2, \ldots$ we write in one of the following ways

$$
\begin{aligned}
q &= \langle e_1, e_2, \ldots \rangle \\
q &= e_1, e_2, \ldots \\
q &= e_1 e_2, \ldots .
\end{aligned}
$$

We denote the empty queue by ϵ.

A.3.1 Length

The length of a queue $q = \langle e_1, e_2, \ldots \rangle$, written $|q|$, is defined as:

$$
|q| \triangleq \begin{cases} n & \text{if } q \text{ is finite and ends in } e_n \\ \infty & \text{if } q \text{ is infinite.} \end{cases}
$$

A.3.2 Head, Tail, Last, Init

If $q = \langle e_1, e_2, e_3, \ldots \rangle$ is nonempty, define

$$
\begin{aligned}
\text{head}(q) &\triangleq e_1 \\
\text{tail}(q) &\triangleq \langle e_2, e_3, \ldots \rangle .
\end{aligned}
$$

If q is finite and ends in e_n, then define

$$last(q) \triangleq e_n$$
$$init(q) \triangleq \langle e_1, e_2, \ldots, e_{n-1} \rangle.$$

A.3.3 Cross product

The cross product of a queue $q = \langle e_1, e_2, \ldots, e_n \rangle$ and a constant c written $q \times c$ returns a queue where each element is paired with c. That is,

$$q \times c \triangleq \langle (e_1, c), (e_2, c), \ldots, (e_n, c) \rangle$$

A.3.4 Concatenation

Concatenation of two queues l_1 and l_2 written $l_1 \cdot l_2$ or sometimes $l_1 l_2$, is defined when l_1 is finite. If $l_1 = \langle e_1, \ldots, e_n \rangle$ and $l_2 = \langle e_{n+1}, e_{n+2} \ldots \rangle$, then define

$$l_1 \cdot l_2 \triangleq \langle e_1, \ldots, e_n, e_{n+1}, e_{n+2} \ldots \rangle$$

A.3.5 Indexing

If $q = \langle e_1, e_2, \ldots \rangle$, then define for all i with $1 \leq i < |q|$

$$q[i] \triangleq e_i$$

We let *dom(q)* denote the set of indices of any queue q. Thus,

$$dom(q) \triangleq \{i | 1 \leq i \leq |q|\}.$$

We denote as *suffixes(q)* the set of sets of consecutive indices at the end of any queue q. Thus,

$$\textit{suffixes(q)} \triangleq \{\{i|j \leq i \leq |q|\}|1 \leq j \leq |q|\}.$$

We denote as *prefixes(q)* the set of sets of consecutive indices at the begininig of any queue q. Thus,

$$\textit{prefixes}(q) \triangleq \{\{i|1 \leq i \leq j\}|0 \leq j \leq |q|\}.$$

If q is nonempty, we denote by *maxindex(q)* the maximum index in q. Thus,

$$\textit{maxindex}(q) \triangleq |q|.$$

The function *delete(q, I)* deletes elements of q with indices in the, possibly empty, set I from *dom(q)*. Thus,

$$\textit{delete}(q, I) \triangleq \ \langle q[i] \mid i \in \textit{dom}(q) \ \wedge \ i \notin I\rangle.$$

A.4 Functions and Mappings

We use the term "function" and "mapping" synonymously. We use standard notation for function definition and application. When explicitly defining the mapping from elements to elements we use notation like

$$\begin{array}{c}[1 \mapsto 2,\\ 2 \mapsto 4,\\ \ldots,\\ 8 \mapsto 16]\end{array}$$

or equivalently $[i \mapsto 2i \mid 1 \leq i \leq 8]$.

A.4.1 Function Type

A function f mapping elements from set A to set B has the type

$$A \rightarrow B$$

We only deal with total functions, that is, $f(a)$ is defined for all elements $a \in A$. A is referred to as the *domain* of f.

A.4.2 Domain and Range

For any function f, *dom(f)* denotes the domain of f. The *range* of f, denoted as *rng(f)*, is defined as

$$rng(f) \triangleq \{f(e) \mid e \in dom(f)\}.$$

A.4.3 Operations of Functions

For a function $f : A \rightarrow B$ and $g : C \rightarrow D$ with $B \subseteq C$, define the *composition* $f \circ g : A \rightarrow D$ such that for all $a \in A$,

$$(f \circ g)(a) = f(g(a)).$$

For any function $f : A \rightarrow B$ and set S, $f \restriction S$ denotes the function of type $(A \cap S) \rightarrow B$ such that for all $a \in A \cap S$,

$$(f \restriction S)(a) = f(a).$$

Appendix B

Invariance proofs for $\mathcal{TCP}^h$

In the proofs of the invariants we use brackets "[ ]" to refer to the value of variables received in a segment from the channels as opposed to the actual value of the variable at the host. For example, if the client gets the input *receive-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)*, we write $[sn_s]$ to refer to the value of the variable sn_s the client gets in the segment. We make this distinction because it is possible that the value of a variable at a host might have changed since it was sent out on a segment. We also use the operators sn and ack to return the sequence number and acknowledgment number respectively of a segment on a channel.

We use the standard inductive technique for proving the invariants. That is, we show that the invariants hold for the start states and then show that for every step (s, a, s') of $\mathcal{TCP}^h$, if the invariant holds in state s then it also holds in state s'. The invariants are typically of the form "if P (premise) then C (consequence)", where P might be true, so we only need to consider actions that could cause P to go from false to true or could cause C to go from true to false. We call these actions critical actions.

Below when we say a proof is symmetric we mean that the actions are the same except with different subscripts, so for example, $crash_s$ is symmetric to $crash_c$ and *send-seg*$_{cs}$*(*sn_c*,* ack_c*,*msg_c*)* is symmetric to *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)*. In the symmetric proofs, state values remain the same, except where indicated.

Invariant 7.1

1. For all segments $p \in$ *in-transit*$_{cs}$, $sn_c \geq sn(p)$.

2. For all segments $p \in \textit{in-transit}_{sc}$, $sn_s \geq sn(p)$.

Proof: In the start states $\textit{in-transit}_{cs}$ and $\textit{in-transit}_{sc}$ are both empty, so the invariant holds for the base case.

1. We now consider the critical steps for Part 1.

   $\underline{a = \textit{send-seg}_{cs}(p)}$

   Any step that places a segment in $\textit{in-transit}_{cs}$ is critical. However, for any such segment the sequence number of the segment is the current sequence number of the client, so Part 1 is not violated.

   $\underline{a = \textit{duplicate}_{cs}(p)}$

   This step also adds segments to $\textit{in-transit}_{cs}$. However, since p must be an exact duplicate of a segment already in $\textit{in-transit}_{cs}$, if Part 1 holds in state s it also holds in state s'.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.2

1. If $ack_s \in \mathtt{N}$ then $ack_s \leq sn_c + 1$.

2. If $ack_c \in \mathtt{N}$ then $ack_c \leq sn_s + 1$.

Proof: In the initial state ack_s and ack_c are both undefined, so the invariant holds for this case.

1. We consider the critical steps for Part 1.

   $\underline{a = \textit{receive-seg}_{cs}(p)}$

   Any step where the client receives a segment p may assign ack_s a non-`nil` value. However, the value assigned is always $sn(p) + 1$. Segment p must have been in $\textit{in-transit}_{cs}$ in state s, and since the step doesn not change the value of $sn(p)$, we know from Invariant 7.1 that $sn_c \geq sn(p)$. Therefore, Part 1 holds.

2. The proof for Part 2 is symmetric. ■

Invariant 7.3

1. For all segments $p \in \mathit{in\text{-}transit}_{sc}$, $\mathit{ack}(p) \leq \mathit{sn}_c + 1$.

2. For all segments $p \in \mathit{in\text{-}transit}_{cs}$, $\mathit{ack}(p) \leq \mathit{sn}_s + 1$.

Proof: In the start states $\mathit{in\text{-}transit}_{cs}$ and $\mathit{in\text{-}transit}_{sc}$ are both empty, so the invariant holds for the base case.

1. We now consider the critical step for Part 1.

   $a = \mathit{send\text{-}seg}_{sc}(p)$

   Any step that places a segment in $\mathit{in\text{-}transit}_{sc}$ is critical. For any such segment p, $\mathit{ack}(p) = \mathit{ack}_s$, and from Invariant 7.2 we know that $\mathit{ack}_s \leq \mathit{sn}_c + 1$. Therefore Part 1 holds after this step.

   $a = \mathit{duplicate}_{sc}(p)$

   This step also adds segments to $\mathit{in\text{-}transit}_{sc}$. However, since p must be an exact duplicate of a segment already in $\mathit{in\text{-}transit}_{sc}$, if Part 1 holds in state s it also holds in state s'.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.4

1. If $\mathit{mode}_c =$ `syn-sent` then for all non-SYN segments $p \in \mathit{in\text{-}transit}_{cs}$, $\mathit{sn}(p) < \mathit{isn}_c$.

2. If $\mathit{mode}_s =$ `syn-rcvd` then for all non-SYN segments $p \in \mathit{in\text{-}transit}_{sc}$, $\mathit{sn}(p) < \mathit{isn}_s$.

Proof: In the start state mode_c and mode_s have the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

1. $a = \mathit{send\text{-}msg}_c(\mathit{open}, m, \mathit{close})$

   This step can make the premise of the invariant go from false to true. However, if this assignment is made, then sn_c gets assigned $s.\mathit{sn}_c + 1$. We know from Invariant 7.1 that $s.\mathit{sn}_c \geq \mathit{sn}(p)$ for any p in $s.\mathit{in\text{-}transit}_{cs}$. Since this step does not add any segments to $\mathit{in\text{-}transit}_{cs}$, we know after this step $s'.\mathit{isn}_c > s.\mathit{sn}_c \geq \mathit{sn}(p)$ for all segments p in $\mathit{in\text{-}transit}_{cs}$.

a = *send-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*)* and a = *send-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*)*

Both these steps can make the consequence of the Part 1 go from true to false. However, both actions are only enabled if $mode_c \neq$ `syn-sent`, and neither changes $mode_c$, so Part 1 holds after either of these steps.

2. a = *send-seg*$_{cs}$*(SYN, *sn_c*)*

   This step is symmetric to a = *send-msg*$_c$*(open, m, close)* for Part 1.

   a = *send-seg*$_{sc}$*(*sn_s*, *ack_s*, *msg_s*)* and a = *send-seg*$_{sc}$*(*sn_s*, *ack_s*, *msg_s*)*

   The proof for these steps is symmetric to the case for the symmetric steps of Part 1.

   ■

Invariant 7.5

1. $isn_c \neq$ `nil` if and only if $mode_c \neq$ `closed`.

2. $isn_s^c \neq$ `nil` if and only if $mode_c \notin \{$`closed`, `syn-sent`$\}$.

3. $isn_s \neq$ `nil` $\vee$ $isn_c^s \neq$ `nil` if and only if $mode_s \notin \{$`closed`, `listen`$\}$.

Proof: In the start state $isn_c =$ `nil`, $isn_s^c =$ `nil`, and $isn_s =$ `nil`, so the invariant holds in the base case. We now consider the critical steps for Part 1.

1. a = *send-msg*$_c$*(open, m, close)*

   This step may change isn_c from `nil` to a non-`nil` value. However, if this change is made, then $mode_c$ also changes to `syn-sent`. This step may also change $mode_c$ to `closed`, but by definition $mode_c$ being `closed` means isn_c is `nil`, so Part 1 holds.

   a = *receive-seg*$_{sc}$*(*sn_s*, *ack_s*, *msg_s*)*, a = *timeout*$_c$, a = *recover*$_c$, and a = *shut-down*$_c$

   These steps may change $mode_c$ to `closed`, which again by definition, means isn_c is `nil`, so Part 1 holds.

2. We consider the critical steps for Part 2.

   a = *receive-seg*$_{sc}$*(SYN, *sn_s*, *ack_s*)*

   This step may change isn_s^c from `nil` to a non-`nil` value. However, if this change is made, then $mode_c$ also changes to `estb`.

The steps with a = *receive-seg$_{sc}$(sn_s, ack_s, msg_s)*, a = *timeout$_c$*, a = *recover$_c$*, and a = *shut-down$_c$* may all change $mode_c$ to `closed`, but by definition $mode_c$ being `closed` means isn_s^c is `nil`.

3. We consider the critical steps for Part 3.

   <u>a = *receive-seg$_{cs}$(SYN, sn_c)*</u>

   This step causes isn_s to go from `nil` to a sn_c and isn_c^s to also go to a non-`nil` value. However, this step also changes $mode_s$ to `syn-rcvd`, so Part 3 holds.

   <u>a = *send-msg$_s$(m, close)*</u>

   This step may change $mode_s$ to `closed`. However, by definition if $mode_s$ is `closed`, then isn_s and isn_c^s are both `nil`.

   <u>a = *receive-seg$_{cs}$(sn_c, ack_c, msg_c)*, a = *timeout$_s$*, a = *recover$_s$*, and a = *shut-down$_s$*</u>

   These steps may change $mode_s$ to `closed`. ■

Invariant 7.6

1. If $isn_c^s \neq$ `nil` then $isn_c^s \leq sn_c$.

2. If $isn_c^s \neq$ `nil` then $isn_c^s < ack_s$.

3. If $isn_s^c \neq$ `nil` then $isn_s^c \leq sn_s$.

4. If $isn_s^c \neq$ `nil` then $isn_s^c < ack_c$.

5. If $isn_c \neq$ `nil` then $isn_c \leq sn_c$.

6. If $isn_s \neq$ `nil` then $isn_s \leq sn_s$.

Proof: In the start state $isn_c^s, isn_s^c, isn_c, isn_s$ are all equal to `nil`, so the invariant holds in this state. We examine the critical steps below.

1. <u>a = *receive-seg$_{cs}$(SYN, sn_c)*</u>

   In this step isn_c^s is assigned $[sn_c]$. For this assignment to happen $[sn_c]$ must have been in $s.in\text{-}transit_{cs}$. Since Invariant 7.1 holds in state s, we know $s.sn_c \geq [sn_c]$. Therefore, $sn_c \geq isn_c^s$.

2. For Part 2 the critical step is:

   $\underline{a = \textit{receive-seg}_{cs}(SYN,\ sn_c)}$

   In this step isn_c^s is assigned $[sn_c]$. However, ack_s is assigned to $[sn_c] + 1$, so Part 2 holds after this step.

3. For Part 3 the critical step is:

   $\underline{a = \textit{receive-seg}_{sc}(SYN,\ sn_s,\ ack_s)}$

   In this step isn_c^s is assigned $[sn_s]$. For this assignment to happen $[sn_s]$ must have been in $s.\textit{in-transit}_{sc}$. Since Invariant 7.1 holds in state s, we know $s.sn_s \geq [sn_s]$. Therefore, $sn_s \geq isn_s^c$.

4. For Part 4 the critical step is:

   $\underline{a = \textit{receive-seg}_{sc}(SYN,\ sn_s,\ ack_s)}$

   In this step isn_s^c is assigned $[sn_s]$. However, ack_c is assigned to $[sn_s] + 1$, so Part 4 holds after this step.

5. For Part 5 the critical step is:

   $\underline{a = \textit{send-msg}_c(open,\ m,\ close)}$

   In this step isn_c is assigned sn_c; therefore, Part 5 holds after this step.

6. For Part 6 the critical step is:

   $\underline{a = \textit{receive-seg}_{cs}(SYN,\ sn_c)}$

   In this step isn_s is assigned sn_s; therefore, Part 6 holds after this step. ∎

Invariant 7.7

If $mode_s =$ `syn-rcvd` then $ack_s = isn_c^s + 1$.

Proof: In the start state $mode_s =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$\underline{a = \textit{receive-seg}_{cs}(SYN,\ sn_c)}$

This step assigns $mode_s$ to `syn-rcvd`, but also assigns ack_s to $[sn_s] + 1$ and isn_c^s to $[sn_s]$, so the invariant holds after this step.

$\underline{a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c)}$

This step may change ack_s, but since we assume the invariant holds in state s, either

$s.mode_s \neq$ `syn-rcvd`, and this step does not change it to `syn-rcvd`, or if $s.mode_s =$ `syn-rcvd`, after this step $s'.mode_s =$ `estb`, so the invariant holds after this step.

<u>a = *receive-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*, FIN)*</u>

The proof is the same as the case for a = *receive-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*)* except that if $s.mode_s =$ `syn-rcvd`, after this step $s'.mode_s =$ `close-wait`. ■

Invariant 7.8

If $(i, j) \in assoc$ then $i \leq sn_c \wedge j \leq sn_s$.

Proof: In the start state *assoc* is the empty set, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

<u>a = *receive-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*)* and a = *receive-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*,FIN)*</u>

These steps may add the pair (isn_c^s, isn_s) to *assoc*. However, from Invariant 7.6 we know that $isn_c^s \leq sn_c$ and $isn_s \leq sn_s$, so the invariant holds after these steps. ■

Invariant 7.9

1. If $isn_c \neq$ `nil` $\wedge$ $choose\text{-}isn_c =$ `true` then $isn_c \neq isn_c^s$.

2. If $isn_s \neq$ `nil` $\wedge$ $choose\text{-}isn_s =$ `true` then $isn_s \neq isn_s^c$.

Proof: In the start state $isn_c =$ `nil` and $isn_s =$ `nil`, so the invariant holds for the base case.

1. We consider the critical steps for Part 1.

   <u>a = *send-msg*$_c$*(open, m, close)*</u>

   In this step isn_c is assigned $s.sn_c + 1$ and $choose\text{-}isn_c$ is assigned `true`. If $s.isn_c^s =$ `nil` then Part 1 holds. If $s.isn_c^s \neq$ `nil`, then from Invariant 7.6 we know $s.isn_c^s \leq s.sn_c$. Since $s'.isn_c = s.sn_c + 1$, and this step does not change isn_c^s, we know $s'.isn_c > s'.isn_c^s$, so Part 1 holds after this step.

   <u>a = *receive-seg*$_{cs}$*(SYN, *sn_c*)*</u>

   In this step isn_c^s is assigned $[sn_c]$, but $choose\text{-}isn_c$ is also assigned `false`, so Part 1 holds after this step.

2. We consider the critical steps for Part 2.

   <u>a = *receive-seg*$_{cs}$*(SYN, *sn_c*)*</u>

In this step isn_s is assigned $s.sn_s+1$, and $choose\text{-}isn_s$ is assigned `true`. If $s.isn_s^c = \mathtt{nil}$ then Part 2 holds. If $s.isn_s^c \neq \mathtt{nil}$, then from Invariant 7.6 we know $s.isn_s^c \leq s.sn_s$. Since $s'.isn_s = s.sn_s+1$, and this step does not change isn_s^c, we know $s'.isn_s > s'.isn_s^c$, so Part 1 holds after this step. ∎

Invariant 7.10

1. If $mode_c = \mathtt{syn\text{-}sent}$ then $sn_c = isn_c$.

2. If $mode_c = \mathtt{syn\text{-}rcvd}$ then $sn_s = isn_s$.

Proof: In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state.

1. We examine the critical steps for Part 1.

   $\underline{a = send\text{-}msg_c(open,\ m,\ close)}$

   This step assigns $mode_c$ to `syn-sent` and increments sn_c, but also assigns isn_c to sn_c. Therefore, Part 1 holds after this step.

   $\underline{a = prepare\text{-}msg_c}$

   This step increments sn_c, but it is only enabled if $mode_c \in \{\mathtt{estb}, \mathtt{close\text{-}wait}\}$. Since we assumed the invariant holds for state s, this step is not enabled in state s.

   $\underline{a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s),\ a = timeout_c,\ a = recover_c,\ \text{and}\ a = shut\text{-}down_c}$

   These steps may change isn_c to `nil`, because they may set $mode_c$ to `closed`, so Part 1 holds after these steps.

2. We consider the critical steps for Part 2

   $\underline{a = receive\text{-}seg_{cs}(SYN,\ sn_c)}$

   This step assigns $mode_s$ to `syn-rcvd` and increments sn_s, but also assigns isn_s to sn_s. Therefore, Part 2 holds after this step.

   $\underline{a = prepare\text{-}msg_s}$

   Symmetric to the case for $a = prepare\text{-}msg_c$ of Part 1.

   $\underline{a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c),\ a = timeout_s,\ a = recover_s,\ \text{and}\ a = shut\text{-}down_s}$

   These steps may change isn_s to `nil`, because they may set $mode_s$ to `closed`, so Part 1 holds after these steps. ∎

Invariant 7.11

1. If *choose-isn*$_c$ $\wedge$ $isn_c = i$ then $\forall$ SYN segments $p \in$ *in-transit*$_{cs}$, $sn(p) < i \wedge$ $\forall$ SYN segments $q \in$ *in-transit*$_{sc}$, $ack(q) < i + 1$.

2. If *choose-isn*$_s \wedge isn_s = i$ then $\forall$ SYN segments $p \in$ *in-transit*$_{sc}$, $sn(p) < j \wedge \forall$ segments $q \in$ *in-transit*$_{cs}$, $ack(q) < i + 1$.

Proof: In the initial state *choose-isn*$_c$ = `false` and *choose-isn*$_s$ = `false` so the invariant holds for the base case. We examine critical steps of the form (s, a, s') for Part 1 below.

1. a = *send-msg*$_c$*(open, m, close)*

   This step can make the premise of Part 1 go from false to true. In this step *choose-isn*$_c$ may get assigned `true` and isn_c may get assigned $s.sn_c$+1. We know from Invariant 7.1 that $s.sn_c \geq sn(p)$ for any p in $s.$*in-transit*$_{cs}$. Since this step does not add any segments to *in-transit*$_{cs}$, we know after this step $s'.isn_c > s.sn_c \geq sn(p)$ for all segments p in *in-transit*$_{cs}$. From Invariant 7.3, we know that for any segment q in $s.$*in-transit*$_{sc}$, $ack(q) \leq s.sn_c + 1$. Since $s'.isn_c = s.sn_c + 1$, we know that for any $q \in s'.$*in-transit*$_{sc}$, $ack(q) < s'.isn_c + 1$. Therefore, Part 1 holds after this step.

   a = *send-seg*$_{cs}$*(SYN,* sn_c*)*

   This step adds a segment to *in-transit*$_{cs}$, but it also sets *choose-isn*$_c$ to `false`, so Part 1 holds after this step.

   a = *send-seg*$_{sc}$*(SYN,* sn_s, ack_s*)*

   This step adds segments to *in-transit*$_{sc}$, but it also sets *choose-isn*$_c$ to `false`, so Part 1 holds after this step.

2. We examine the critical steps for Part 2 below.

   a = *receive-seg*$_{cs}$*(SYN,* sn_c*)*

   The proof for this case is symmetric to the proof for a = *send-msg*$_c$*(open, m, close)* of Part 1.

   a = *send-seg*$_{sc}$*(SYN,* sn_s, ack_s*)*

   This step sets *choose-isn*$_s$ to `false`.

   a = *send-seg*$_{cs}$*(*sn_c, ack_c, msg_c*)* and a = *send-seg*$_{cs}$*(*sn_c, ack_c, msg_c*, FIN)*

   These steps set *choose-isn*$_s$ to `false`. ■

Invariant 7.12

1. For all $i \in \mathtt{N} \cup \{\mathtt{nil}\}$, $(i, \mathtt{nil}) \notin$ *estb-pairs*.

2. For all $j \in \mathtt{N} \cup \{\mathtt{nil}\}$, $(\mathtt{nil}, j) \notin$ *estb-pairs*.

3. For all $i \in \mathtt{N} \cup \{\mathtt{nil}\}$, $(i, \mathtt{nil}) \notin$ *assoc*.

4. For all $j \in \mathtt{N} \cup \{\mathtt{nil}\}$, $(\mathtt{nil}, j) \notin$ *assoc*.

Proof: . In the start state *estb-pairs* and *assoc* are both the empty set. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = \textit{receive-seg}_{sc}(SYN,\ sn_s,\ ack_s)}$
   In this step (isn_c, sn_s) gets added to *estb-pairs*, if $mode_c = \mathtt{syn\text{-}sent}$. However, from Invariant 7.5 we know that if $mode_c \neq \mathtt{closed}$ then $isn_c \neq \mathtt{nil}$, so Part 1 holds after this step.

2. The proof for Part 2 is the same as the proof for Part 1.

3. $\underline{a = \textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c) \text{ and } a = \textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$
   These steps may add (isn_c^s, isn_s) to *assoc*, but from Invariant 7.5 we know neither element of the pair is nil, so Part 3 holds after these steps.

4. The proof is the same as for Part 3. ∎

Invariant 7.13

1. If $mode_c \in \{\mathtt{fin\text{-}wait\text{-}1}, \mathtt{fin\text{-}wait\text{-}2}, \mathtt{closing}, \mathtt{timed\text{-}wait}, \mathtt{last\text{-}ack}\}$ then $\textit{send-buf}_c = \epsilon \wedge \textit{rcvd-close}_c = \mathtt{true}$.

2. If $mode_s \in \{\mathtt{fin\text{-}wait\text{-}1}, \mathtt{fin\text{-}wait\text{-}2}, \mathtt{closing}, \mathtt{timed\text{-}wait}, \mathtt{last\text{-}ack}\}$ then $\textit{send-buf}_s = \epsilon \wedge \textit{rcvd-close}_s = \mathtt{true}$.

Proof: In the start state $mode_c$ and $mode_s$ have the value closed, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. *a = send-msg$_c$(open, m, close)*

   This step can change the consequence of Part 1 from true to false by adding a message to *send-buf*$_c$. However, the message is added only if *mode*$_c \in \{$`syn-sent`, `estb`, `close-wait`$\}$, so Part 1 holds after this step.

   Steps that cause the client to close also change the consequence of Part 1 from true to false. However, the premise of Part 1 is also obviously false after any of these steps.

   a = prepare-msg$_c$

   This step may cause the premise of Part 1 to go from false to true by changing mode$_c$ to `fin-wait-1` or `last-ack`. However, the change is made only if *send-buf*$_c$ = $\epsilon \wedge$ *rcvd-close*$_c$ = `true`. Therefore, Part 1 holds after this step.

2. *a = send-msg$_s$(m, close)*

   This step can change the consequence of Part 2 from true to false by adding a message to *send-buf*$_s$. However, the message is added only if *mode*$_c \in \{$`syn-rcvd`, `estb`, `close-wait`$\}$, so Part 2 holds after this step.

   Steps that cause the server to close also change the consequence of Part 2 from true to false. However, the premise of Part 2 is also obviously false after any of these steps.

   a = prepare-msg$_s$

   Symmetric to the case for a = *prepare-msg*$_c$ for Part 1. ■

Invariant 7.14

If *mode*$_s \in \{$`listen`, `syn-rcvd`$\}$ then *rcv-buf*$_s = \epsilon$.

Proof: In the start state *mode*$_s$ = `closed` so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

a = passive-open

This step may cause the premise of the invariant to go from false to true. However, in this step *rcv-buf*$_s$ is initialized to ϵ, so the invariant holds after this step.

a = receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$) and *a = receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$, FIN)*

These steps may cause the consequence of the invariant to go from true to false. However, after these steps *mode*$_s \notin \{$`listen`, `syn-rcvd`$\}$, so the invariant holds. ■

Invariant 7.15

If $isn_c = isn_c^s$ and there exists $p \in$ *in-transit*$_{cs}$ such that $ack(p) > sn_s$ then $mode_c \neq$ `syn-sent`.

Proof: In the start state *in-transit*$_{cs}$ is empty, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below, but before we do, we point out that $a =$ *receive-seg*$_{cs}$*(SYN,* sn_c*)* is not a critical step, because Invariant 7.11 tells us that after this step there are no segments $p \in$ *in-transit*$_{cs}$ such that $ack(p) > sn_s$.

<u>a = *send-msg*$_c$*(open, m, close)*</u>

This step may change the consequence of the invariant from true to false, but from Invariant 7.9 we know that if this happens $isn_c \neq isn_c^s$. Therefore the invariant holds after this step.

<u>a = *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and a = *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*</u>

These steps add a segment to *in-transit*$_{cs}$, but requires $mode_c \neq$ `syn-sent`. ■

Invariant 7.16

1. If $mode_s =$ `syn-rcvd` then for all i, $(i, isn_s) \notin$ *assoc*.
2. If $mode_c =$ `syn-sent` then for all j, $(isn_c, j) \notin$ *assoc*.
3. If $mode_s =$ `syn-rcvd` $\wedge$ *choose-isn*$_s$ then for all i, $(i, isn_s) \notin$ *estb-pairs*.
4. If $mode_c =$ `syn-sent` then for all j, $(isn_c, j) \notin$ *estb-pairs*.

Proof: In the start state $mode_s =$ `closed` and $mode_c =$ `closed`, so the invariant holds in this state. We consider the critical steps below.

1. <u>a = *receive-seg*$_{cs}$*(SYN,* sn_c*)*</u>

   This step changes $mode_s$ to `syn-rcvd` and assigns isn_s to $s.sn_s{+}1$. Since this step does not add any elements to *assoc*, if there exists i, such that $(i, isn_s) \in$ *assoc*, then the pair must be in *assoc* in state s. From Invariant 7.8 we know that if $(i, isn_s) \in$ *assoc* then $isn_s \leq s.sn_s$. However, after this step $isn_s = s.sn_s + 1$. Therefore, we know that there cannot exist an i such that $(i, isn_s) \in$ *assoc*.

   <u>a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)*</u>

   This step may add the pair (isn_c^s, isn_s) to *assoc*. However, if the pair is added, $mode_s$

is also assigned `estb`.

<u>a = *receive-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*, FIN)*</u>

This step may add the pair (isn_c^s, isn_s) to *assoc*. However, if the pair is added, $mode_s$ is also assigned `close-wait`.

2. We consider the critical steps for Part 2 below.

   <u>a = *send-msg*$_c$*(open, m, close)*</u>

   This step changes $mode_c$ to `syn-sent` and assigns isn_c to $s.sn_c{+}1$. Since this step does not add any elements to *assoc*, if there exists j, such that $(isn_c, j) \in assoc$, then the pair must be in *assoc* in state s. From Invariant 7.8 we know that if $(isn_c, j) \in s.assoc$ then $isn_c \leq s.sn_c$. However, after this step $isn_c = s.sn_c + 1$. Therefore, we know that there cannot exist an j such that $(isn_c, j) \in s.assoc$.

   <u>a = *receive-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*)* and a = *receive-seg*$_{cs}$*(*sn_c*, *ack_c*, *msg_c*, FIN)*</u>

   If $[ack_c] = sn_s + 1$, these steps may add the pair (isn_c^s, isn_s) to *assoc*. Thus, if $isn_c^s = isn_c$, the consequence of the invariant goes from true to false. However, Invariant 7.15 tells us that if $isn_c^s = isn_c$ and there exists $p \in s.in\text{-}transit_{cs}$ such that $ack(p) > sn_s$, then $s.mode_c \neq$ `syn-sent`. Since these steps do not change the value of $mode_c$, Part 2 holds after these steps.

3. <u>a = *receive-seg*$_{cs}$*(SYN, *sn_c*)*</u>

   This step changes $mode_s$ to `syn-rcvd`, assigns $choose\text{-}isn_s$ to `true`, and assigns isn_s to $s.sn_s + 1$. Clearly after this step $(i, isn_s) \notin estb\text{-}pairs$, so Part 3 holds.

   <u>a = *receive-seg*$_{sc}$*(SYN, *sn_s*, *ack_s*)*</u>

   This step adds a pair to *estb-pairs*, but also sets $choose\text{-}isn_s$ to `false`, so Part 3 holds after this step.

4. <u>a = *send-msg*$_c$*(open, m, close)*</u>

   This step changes $mode_c$ to `syn-sent`, assigns $choose\text{-}isn_c$ to `true`, and assigns isn_c to $s.sn_c + 1$. Clearly after this step $(isn_c, j) \notin estb\text{-}pairs$, so Part 4 holds.

   <u>a = *receive-seg*$_{sc}$*(SYN, *sn_s*, *ack_s*)*</u>

   This step adds a pair to *estb-pairs*, but also sets $mode_c$ to `estb`. ■

Invariant 7.18

If $(isn_c, isn_s) \in assoc\ \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ then $(isn_c, isn_s^c) \in estb\text{-}pairs$.

Proof: We examine the critical steps.

$a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$

In these steps the pair (isn_c^s, isn_s) may be added to *assoc*. These steps make the premise of the invariant true, if $s.isn_c = s.isn_c^s$, $[ack_c] = s.sn_s + 1$, and $s.mode_s = \texttt{syn-rcvd}$. Thus, by Invariant 7.15 we know $s.mode_c \neq \texttt{syn-sent}$, and by Invariant 7.5 we know $s.isn_c^s \neq \texttt{nil}$, which means $isn_c \neq \texttt{nil}$. Thus, again by Invariant 7.5, we know $s.mode_c \neq \texttt{closed}$. Since the premise of the invariant has $mode_c \notin \{\texttt{rec}, \texttt{reset}\}$, the only other possible values for $mode_c$ is a value in *sync-states*. From Invariant 7.17 we know that if $mode_c \in sync\text{-}states$ then $(isn_c, isn_s^c) \in estb\text{-}pairs$. Therefore, the invariant holds after these steps. ■

Invariant 7.17

1. If $mode_c \in sync\text{-}states$ then $(isn_c, isn_s^c) \in estb\text{-}pairs$

2. If $(isn_c, isn_s^c) \in estb\text{-}pairs \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ then $mode_c \in sync\text{-}states$.

Proof: In the start state $mode_c = \texttt{closed}$ and *estb-pairs* is empty, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

1. $a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$

   In this step $mode_c$ is assigned `estb`, but (isn_c, isn_s^c) also gets added to *estb-pairs*, so Part 1 holds after this step.

2. $a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$

   In this step (isn_c, isn_s^c) gets added to *estb-pairs*, but $mode_c$ is assigned `estb`, so Part 2 holds after this step.

   The steps that cause the client to go to either mode `closed`, `rec`, or `reset`, may change the consequence of Part 2 from true to false, but they also change the premise to false, so Part 2 holds after these steps. ■

Invariant 7.20

If $isn_c = isn_c^s \wedge isn_s = isn_s^c \wedge mode_s = \texttt{syn-rcvd} \wedge mode_c \notin \{\texttt{closed}, \texttt{rec}, \texttt{reset}\}$ then

$mode_c \in$ *sync-states*.

Proof: In the start state $mode_c =$ `closed`, so the invariant holds. Before we consider the critical steps for this invariant, we point out that $a =$ *receive-seg*$_{cs}$*(SYN, sn*$_c$*)* is not critical, because after this step $mode_s =$ `syn-rcvd` and *choose-isn*$_s =$ `true`, which by Invariant 7.9 means $isn_c \neq isn_c^s$. We consider critical steps of the form (s, a, s') below.

$a =$ *receive-seg*$_{sc}$*(SYN, sn*$_s$*, ack*$_s$*)*

This step assigns isn_s^c to $[sn_s]$. However, it also assigns $mode_c$ to `estb`, so the invariant holds after this step.

The steps that can take $mode_c$ out of *sync-states* are all critical, but when they do they add $mode_c$ to the set $\{$`closed`, `rec`, `reset`$\}$, so it is obvious that the invariant holds after these steps, therefore, we do list all these steps here. ■

Invariant 7.19

If $mode_c =$ `syn-sent` $\wedge\ isn_c = isn_c^s$ then $mode_s \notin$ *sync-states*.

Proof: In the start state $mode_c$ has the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a =$ *receive-seg*$_{cs}$*(SYN, sn*$_c$*)*

This step may cause the premise of the invariant to go from false to true, but after this step $mode_s =$ `syn-rcvd`, so the invariant holds.

$a =$ *receive-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* and $a =$ *receive-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)*

These steps can make the consequence of the invariant go from true to false if $[ack_c] = sn_s + 1$. However, if this condition is true, then we know from Invariant 7.15 that $mode_c \neq$ `syn-sent`, so the invariant holds after these steps. ■

Invariant 7.21

If $mode_c =$ `syn-sent` $\wedge\ mode_s =$ `syn-rcvd` $\wedge\ ack_s = sn_c + 1$ then for all segments $p \in$ *in-transit*$_{cs}$, $sn_s \geq ack(p)$.

Proof: In the start state $mode_s =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a =$ *receive-seg*$_{cs}$*(SYN, sn*$_c$*)*

This step assigns $mode_s$ to `syn-rcvd`, ack_s to $[sn_s] + 1$, and sn_s to $s.sn_s + 1$. From Invariant 7.3 we know that for all $p \in s.\mathit{in\text{-}transit}_{cs}$ $\mathit{ack}(p) \leq s.sn_s + 1$. Therefore, after this step, $s'.sn_s \geq \mathit{ack}(p)$.

$a = \mathit{send\text{-}seg}_{cs}(sn_c,\ \mathit{ack}_c,\ \mathit{msg}_c)$ and $a = \mathit{send\text{-}seg}_{cs}(sn_c,\ \mathit{ack}_c,\ \mathit{msg}_c,\ \mathit{FIN})$

These steps can change the consequence of the invariant from true to false by adding a segment p to $\mathit{in\text{-}transit}_{cs}$ with $\mathit{ack}(p) > sn_s$. However, the actions are only enabled if $s.\mathit{mode}_c \neq$ `syn-sent`, and they do not change the value of mode_c, so the invariant holds after either of these steps. ■

Invariant 7.22

If $\mathit{mode}_c =$ `syn-sent` then for all SYN segments $p \in \mathit{in\text{-}transit}_{sc}$ such that $\mathit{ack}(p) = sn_c + 1$, $sn(p) \geq \mathit{ack}(q)$ for all $q \in \mathit{in\text{-}transit}_{cs}$.

Proof: In the start state $\mathit{mode}_s =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a = \mathit{send\text{-}msg}_c(\mathit{open},\ m,\ \mathit{close})$

This step assigns mode_c to `syn-sent`, but we know from Invariant 7.11 that when this assignment is made that there are no SYN segments $p \in \mathit{in\text{-}transit}_{sc}$ such that $\mathit{ack}(p) = sn_c + 1$, so the invariant holds after this step.

$a = \mathit{send\text{-}seg}_{sc}(\mathit{SYN},\ sn_s,\ \mathit{ack}_s)$

This step adds a SYN segment to $\mathit{in\text{-}transit}_{sc}$, so it can change the consequence of the invariant from true to false if the added segment has $\mathit{ack}(p) = sn_c + 1$, but $sn(p) < \mathit{ack}(q)$ for some segment $q \in \mathit{in\text{-}transit}_{cs}$. This action is only enabled if $\mathit{mode}_s =$ `syn-rcvd`. Therefore, we know from Invariant 7.21 that if this steps adds a segment that causes the consequence of the invariant to be false, then $\mathit{mode}_c \neq$ `syn-sent`.

$a = \mathit{send\text{-}seg}_{cs}(sn_c,\ \mathit{ack}_c,\ \mathit{msg}_c)$ or $a = \mathit{send\text{-}seg}_{cs}(sn_c,\ \mathit{ack}_c,\ \mathit{msg}_c,\ \mathit{FIN})$

These steps can change the consequence of the invariant from true to false by adding a segment q to $\mathit{in\text{-}transit}_{cs}$ with $\mathit{ack}(q) > sn(p)$. However, the actions are only enabled if $s.\mathit{mode}_c \neq$ `syn-sent`, and they do not change the value of mode_c, so the invariant holds after either of these steps. ■

Invariant 7.23

If $\mathit{ack}_c \in \mathbb{N}$ then for all $p \in \mathit{in\text{-}transit}_{cs}$, $\mathit{ack}_c \geq \mathit{ack}(p)$.

Proof: In the start state ack_c is undefined, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We point out that a = *receive-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and a = *receive-seg*$_{cs}$*(*sn_s*,* ack_s*,* msg_s*, FIN)* are not critical because they increment ack_c.

a = *send-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*

This step may change the premise of the invariant from false to true by assigning ack_c to $[sn_s] + 1$ if $[ack_s] = sn_c + 1$ and $mode_c = \texttt{syn-sent}$. However, we know from Invariant 7.22 that $[sn_s] \geq ack(p)$ for all $p \in in\text{-}transit_{cs}$, so the invariant holds after this step.

Steps that cause the client to close are also critical, but they also make ack_c undefined, so the invariant holds after any of these steps. ■

Invariant 7.24

If $isn_c = isn_c^s \wedge isn_s = isn_s^c \wedge mode_c \in sync\text{-}states \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ then for all segments $p \in in\text{-}transit_{sc}$, $ack_s \geq ack(p)$.

Proof: In the initial state $mode_c = \texttt{closed}$, so the invariant holds in this state. We examine the critical steps of the form (s, a, s') below.

a = *receive-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*

If $s.mode_c = \texttt{syn-sent} \wedge [ack_s] = s.sn_c + 1$, this step assigns isn_s^c to $[sn_s]$, ack_c to $[sn_s] + 1$, and $mode_c$ to $\texttt{estb}$. From Invariant 7.10 we know that since $s.mode_c = \texttt{syn-sent}$, then $s.isn_c = s.sn_c$. We also know from Invariant 7.6 that $ack_s > isn_c^s = isn_c$. Since by Invariant 7.3 we know that for all $p \in in\text{-}transit_{sc}$, $ack(p) \leq sn_c + 1$, we know that $ack_s \geq ack(p)$ for all such p. Thus, the invariant holds after this step.

a = *receive-seg*$_{cs}$*(SYN,* sn_c*)*

This step assigns ack_s to $[sn_c] + 1$, so may change the consequence of the invariant from true to false. However, this step also assigns isn_s to a non-$\texttt{nil}$ value and assigns $choose\text{-}isn_s$ to $\texttt{true}$. Therefore, from Invariant 7.9, we know $isn_s = isn_s^c$, so the invariant holds after this step. ■

Invariant 7.25

If $mode_c \in sync\text{-}states \wedge mode_s \notin \{\texttt{closed}, \texttt{rec}, \texttt{reset}\} \wedge isn_c = isn_c^s$ and there exists a non-SYN segment $p \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, then $mode_s \neq \texttt{syn-rcvd} \vee ack_c < isn_s$.

Proof: In the start state $mode_c$ has the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a = send\text{-}seg_{cs}(SYN,\ sn_c)$

This step may make the premise of the invariant go from false to true by assigning isn_c^s to isn_c. This step also assigns $mode_s$ to `syn-rcvd`, and increments sn_s and assigns it to isn_s. From Invariant 7.4 we know that for all non-SYN segments $p \in transit_{sc}$ $sn(p) < isn_s$. Therefore, if there exists a non-SYN segment $p \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, which must be the case if the premise of the invariant is true, then after this step $ack_c < isn_s$, so the invariant holds.

$a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$

This step may cause the premise of the invariant to go from false to true if $[ack_s] = sn_c + 1$. Given that $mode_s \notin \{\texttt{closed}, \texttt{rec}, \texttt{reset}\}$, Invariant 7.19 tells us that $mode_s$ must be `syn-rcvd`. Thus, if there exists a non-SYN $p \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, we know that $ack_c < isn_s$ because Invariant 7.4 tells us that $sn(p) < isn_s$. Therefore, the invariant holds after this step.

$a = send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$ and $a = send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$

These steps may also cause the premise of the invariant to go from false to true. However, they are only enabled if $mode_s \neq$ `syn-rcvd`.

$a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$ and $a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$

These steps may change the consequence of the invariant from true to false if $s.mode_s =$ `syn-rcvd`, and the steps assign ack_c to a value greater than or equal to isn_s. However, from Invariant 7.4, we know that if $mode_s =$ `syn-rcvd`, then $isn_s > sn(p)$ for any non-SYN segment $p \in in\text{-}transit_{sc}$. Therefore, the premise of the invariant must also be false if these steps cause the consequence of the invariant to become false. ∎

Invariant 7.26

If $mode_s = \texttt{syn-rcvd} \wedge isn_c = isn_c^s \wedge ack_c = isn_s + 1$ then $isn_s = isn_s^c$.

Proof: In the start state $mode_s$ has the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. The steps with $a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$ and $a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)$ even though they may change ack_c, are not critical. They are not critical because they change ack_c only if $mode_c \in sync\text{-}states$ and

$sn(p) \geq ack_c$, so Invariant 7.25 tells us the after these steps either $mode_s \neq$ `syn-rcvd` $\vee$ $ack_c \neq isn_s + 1$.

<u>*a = receive-seg$_{cs}$(SYN, sn$_c$)*</u>

This step may cause the consequence of the invariant to go from true to false. However, from Invariant 7.2 we know that after this step $ack_c < isn_s + 1$, so the invariant holds after this step.

<u>*a = receive-seg$_{sc}$(SYN, sn$_s$, ack$_s$)*</u>

This step assigns ack_c the value $[sn_s] + 1$ and isn_s^c the value $[sn_s]$. Thus, if $ack_c = sn_s + 1$ after this step, then clearly $isn_s = isn_s^c$. If $isn_s \neq isn_s^c$ after this step, then clearly the premise of the invariant is also false. ∎

Invariant 7.27

If $mode_s =$ `syn-rcvd` $\wedge$ $isn_c = isn_c^s$ and there exists a non-SYN segment $p \in in\text{-}transit_{cs}$ such that $ack(p) = isn_s + 1$ then $isn_s = isn_s^c$.

Proof: In the start state $mode_s$ has the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

<u>*a = receive-seg$_{cs}$(SYN, sn$_c$)*</u>

This step is critical because it can cause the consequence of the invariant to go from true to false, by assigning isn_c^s a value that is not equal to isn_c. However, from Invariant 7.2 we know that after this step $ack_c < isn_s + 1$, and since Invariant 7.23 tells us that for all $p \in in\text{-}transit_{cs}$ $ack_c \geq ack(p)$, we know that the premise of the invariant also becomes false after this step if the consequence becomes false.

<u>*a = receive-seg$_{sc}$(SYN, sn$_s$, ack$_s$)*</u>

This step assigns ack_c the value $[sn_s] + 1$ and isn_s^c the value $[sn_s]$. Thus, it can make the consequence of the invariant false if $[sn_s] \neq isn_s$. If $mode_s =$ `syn-rcvd`, then Invariant 7.10 tells us that $sn_s = isn_s$. We also know from Invariant 7.1 that $sn_s \geq [sn_s]$. Therefore, if $[sn_s] \neq isn_s$, then $[sn_s] < isn_s$, which means $ack_c < isn_s + 1$. From Invariant 7.23 we know that for all segments $p \in transit_{cs}$, $ack_c \geq ack(p)$. Therefore, after this step there cannot be a segment $p \in transit_{cs}$ with $ack(p) = isn_s + 1$. Therefore, the invariant holds after this step.

<u>*a = send-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$)* and *a = send-seg$_{sc}$(sn$_c$, ack$_c$, msg$_c$, FIN)*</u>

These steps may also cause the premise of the invariant to go from false to true by adding a non-SYN segment p to $in\text{-}transit_{cs}$ with $sn(p) = isn_s + 1$. For these steps to have this affect, it must be that in state s, $ack_c = isn_s + 1$. Thus, by Invariant 7.26, $s.isn_s = s.isn_s^c$. Since these steps do not change either isn_s or isn_s^c, the invariant holds after the steps. ■

Invariant 7.28

1. If $(isn_c, isn_s) \in assoc$ then $isn_s^c = isn_s \wedge isn_c^s = isn_c$.

2. If $(isn_c, isn_s) \in estb\text{-}pairs$ then $isn_s^c = isn_s$

Proof: In the start state *assoc* and *estb-pairs* are both the empty set, so the invariant holds in this state. We consider critcal steps of the form (s, a, a') below.

1. $a = receive\text{-}seg_{cs}(SYN,\ sn_c)$

   This step is critical because it can cause the consequence of the invariant to go from true to false, by assigning isn_c^s a value that is not equal to isn_c. However, Invariant 7.16 tells us that in state s, $(isn_c, isn_s) \notin assoc$, and since the step does not add any pairs to *assoc*, we know $(isn_c, isn_s) \notin assoc$ after the step.

   $a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$

   This step is critical because it can cause the consequence of the invariant to go from true to false, by assigning isn_s^c a value that is not equal to isn_s. However, again by Invariant 7.16 we know $(isn_c, isn_s) \notin assoc$ after the step, so Part 1 holds.

   $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$

   These steps can cause the premise of Part 1 to go from false to true by adding the pair (isn_c^s, isn_s) to *assoc* if $s.mode_s =$ `syn-rcvd` and if $[ack_c] = sn_s + 1$. If this assignment is made then clearly $isn_c^s = isn_c$. From Invariant 7.27 we know that $s.isn_s^c = s.isn_s$. Since neither of these variables changes in these steps, Part 1 holds after thesse steps.

   All the steps that cause the client or server to close can also make the consequence of Part 1 go from true to false. However, those steps also make isn_c or isn_s go to `nil`, so by Invariant 7.12 the premise of Part 1 is also false after any of these steps.

2. $a = receive\text{-}seg_{cs}(SYN,\ sn_c)$

   This step is critical because it can cause the consequence of the invariant to go from true to false, by assigning isn_c^s a value that is not equal to isn_c. Since it also assigns $choose\text{-}isn_s$ to `true`, Invariant 7.16 tells us that in state s, $(isn_c, isn_s) \notin estb\text{-}pairs$, and since the step does not add any pairs to $estb\text{-}pairs$, we know $(isn_c, isn_s) \notin estb\text{-}pairs$ after the step.

$a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$

This step can cause the premise of Part 2 to go from false to true, and it can cause the consequence of Part 2 to go from true to false. For either case it is clear that Part 2 holds after this step.

All the steps that cause the client or server to close can also make the consequence of Part 2 go from true to false. As for Part 1, we know Part 2 holds after these steps. ■

Invariant 7.29

If $(isn_c, isn_s) \in assoc \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ then for all segments $p \in in\text{-}transit_{sc}$, $ack_s \geq ack(p)$.

Proof: In the initial state $assoc$ is the empty set, so the invariant holds in this state. We examine the critical steps of the form (s, a, s') below.

$a = receive\text{-}seg_{cs}(SYN,\ sn_c)$

This step may make the consequence of the invariant go from true to false, but after this step $mode_s = \texttt{syn-rcvd}$, and from Invariant 7.16 we know that $(isn_c, isn_s) \notin assoc$. Therefore, the premise of the invariant is also false after this step.

$a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$

These steps add the pair (isn_c^s, isn_s) to $assoc$ if $s.mode_s = \texttt{syn-rcvd}$. Thus, it can make the premise of the invariant go from false to true. If the premise of the invariant is true, Invariant 7.28 tells us that $isn_s^c = isn_s \wedge isn_c^s = isn_c$. We also know from Invariants 7.17 and 7.18 that if the premise is true, $mode_c \in sync\text{-}states$. Therefore, by Invariant 7.24 we know that for all segments $p \in s.in\text{-}transit_{sc}$, $s.ack_s \geq ack(p)$. Since these steps do not change the elements of $in\text{-}transit_{sc}$, the invariant holds after these steps. ■

Invariant 7.30

1. If $mode_c \in \{\texttt{estb}, \texttt{close-wait}\} \wedge \neg ready\text{-}to\text{-}send_c \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in estb\text{-}pairs \wedge isn_c^s = isn_c$ then $sn_c < ack_s$.

2. If $mode_s \in \{\texttt{estb}, \texttt{close-wait}\} \wedge \neg \mathit{ready\text{-}to\text{-}send}_s \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in$ $assoc$ then $sn_s < ack_c$.

Proof: In the initial state $mode_c$ and $mode_s$ are `closed`, so the invariant holds in this state. We consider the critical steps of the form (s, a, s') for Part 1 below.

1. <u>*a = send-msg*$_c$*(open, m, close)*</u>

   This step may cause the consequence of Part 1 to go from true to false by incrementing sn_c, but if it does, it also assigns $mode_c$ to `syn-sent`, so Part 1 holds after this step.

   <u>*a = prepare-msg*$_c$</u>

   This step is critical because it increments sn_c and may change $\mathit{ready\text{-}to\text{-}send}_c$ to `false`. However, either sn_c is incremented, and $\mathit{ready\text{-}to\text{-}send}_c$ is set to `true`, or sn_c is incremented, $\mathit{ready\text{-}to\text{-}send}_c$ is set to `false`, and $mode_c$ is set to `fin-wait-1` or `last-ack`. In either case Part 1 still holds after this step.

   <u>*a = receive-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*</u>

   This step changes $mode_c$ to `estb`, adds (isn_c, isn_s) to *estb-pairs*, and sets $\mathit{ready\text{-}to\text{-}send}_c$ to `false`. However, these changes are only made if $[ack_s] = sn_c + 1$. Since Invariant 7.24 tells us that $[ack_s] \leq ack_s$, we know Part 1 holds after this step.

   <u>*a = receive-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and *a = receive-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)*</u>

   These steps may change $\mathit{ready\text{-}to\text{-}send}_c$ to `false`. However, this change is only made if $[ack_s] = sn_c + 1$ and $mode_c \in \mathit{sync\text{-}states}$. Since by Invariant 7.24 we know that $[ack_s] \leq ack_s$, Part 1 holds after these steps.

2. We now consider the critical steps for Part 2.

   <u>*a = receive-seg*$_{cs}$*(SYN,* sn_c*)*</u>

   This step increments sn_s, but it also sets $mode_s$ to `syn-rcvd`, so Part 2 holds after this step.

   <u>*a = prepare-msg*$_s$</u>

   The proof that Part 2 holds after this step is symmetric to the proof that Part 1 holds after the *(s, prepare-msg*$_c$*, s')* step.

   <u>*a = receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and *a = receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*</u>

These steps may change $\textit{ready-to-send}_c$ to `false`. However, this change is only made if $[ack_s] = sn_c + 1$. From Invariants 7.18 and 7.17, we know that $mode_c \neq$ `syn-sent`, so we know $ack_c \in \mathtt{N}$. Therefore, by Invariant 7.23 we know that $[ack_c] \leq ack_c$, so Part 2 holds after this step. ■

Invariant 7.31

1. If $(i, isn_s) \in assoc$ then $isn_c^s = i$.

2. If $(isn_c^s, j) \in assoc \wedge mode_s \in \textit{sync-states}$ then $isn_s = j$.

Proof: In the start state *assoc* is the empty set, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

1. <u>$a = \textit{receive-seg}_{cs}(SYN,\ sn_c)$</u>

   This step may assign isn_c^s a value other than i and isn_s the value j. However, if these assignments are made, $mode_s$ is also assigned the value `syn-rcvd`. Thus, from Invariant 7.16 we know that $(i, j) \notin assoc$.

   <u>$a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$</u>

   These steps may add the pair (isn_c^s, isn_s) to *assoc*, so Part 1 clearly holds after these steps.

2. <u>$a = \textit{receive-seg}_{cs}(SYN,\ sn_c)$</u>

   This step may assign isn_s a value other than j. However, this step also assigns $mode_s$ the value `syn-rcvd`, so Part 2 holds after this step.

   <u>$a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$</u>

   These steps may change the premise of Part 2 from false to true by adding (isn_c^s, isn_s) to *assoc*. However, the consequence is also obviously true after the pair is added, so Part 2 holds after these steps. ■

Invariant 7.32

If $(isn_c^s, j) \in assoc \wedge isn_s \neq j \wedge mode_s \notin \{\mathtt{rec}, \mathtt{reset}\}$ then $mode_s =$ `syn-rcvd`.

Proof: In the start state *assoc* is the empty set, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$\underline{a = \textit{receive-seg}_{cs}(SYN,\ sn_c)}$

This step may assign isn_c^s the value i and isn_s a value other than j. However it also assigns $mode_s$ to `syn-rcvd`.

$\underline{a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c) \text{ and } a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$

These steps may change the consequence of the invariant from true to false by assigning $mode_s$ to `estb` or to `close-wait`. If this change is made the pair (isn_c^s, isn_s) is also added to *assoc*. However, we know from Invariant 7.31 that if $(isn_c^s, j) \in assoc \wedge mode_s \in \textit{sync-states}$, then $isn_s = j$. Therefore, the premise, of the invariant is also false after these steps, so the invariant holds. ∎

Invariant 7.33

1. If $(h, j) \in assoc \wedge (i, j) \in assoc$ then $h = i$.

2. If $(i, j) \in assoc \wedge (i, k) \in assoc$ then $j = k$.

Proof: In the start state *assoc* is the empty set, so the invariant holds in this state. We consider critical steps of the form (s, a, s') for Part 1 below.

1. $\underline{a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c) \text{ and } a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$

   These steps may add the pair (isn_c^s, isn_s) to *assoc*. Without loss of generality assume the pair $(h, j) \in s.assoc$ then these steps can make the premise of Part 1 go from false to true if $s.isn_c^s = i$ and $s.isn_s = j$. However, from Invariant 7.31 we know that if $(h, j) \in s.assoc \wedge isn_s = j$ then $s.isn_c^s = h$. Therefore, $h = i$, so Part 1 holds after these steps.

2. We next consider the critical steps for Part 2.

   $\underline{a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c) \text{ and } a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$

   These steps may add the pair (isn_c^s, isn_s) to *assoc*. Without loss of generality assume the pair $(i, j) \in s.assoc$ then these steps can make the premise of Part 2 go from false to true if $s.isn_c^s = i$ and $s.isn_s = k$. If the pair is added, then after these steps $s'.mode_s \in \{\texttt{estb}, \texttt{close-wait}\}$, and since the step does not change isn_s, $s'.isn_s = k$. From Invariant 7.32, we know that if $(i, j) \in assoc \wedge isn_c^s = i \wedge isn_s \neq j \wedge mode_s \notin$

$\{\texttt{rec}, \texttt{reset}\}$, then $mode_s = \texttt{syn-rcvd}$. Since $mode_s \neq \texttt{syn-rcvd}$, the premise of Invariant 7.32 must be false. The only clause in the premise that can be false is $isn_s \neq j$. Therefore $isn_s = j$, which means $k = j$. Therefore, Part 2 holds after these steps. ■

Invariant 7.34

If $mode_s \in \{\texttt{syn-rcvd}\} \cup \textit{sync-states} \ \wedge mode_c \in \textit{sync-states}$ and there exists i such that $(i, isn_s) \in \textit{estb-pairs}$ then $i = isn_c$.

Proof: In the start state *estb-pairs* is the empty set, so the invariant holds in this state. We consider critcal steps of the form (s, a, a') below.

<u>a = *send-msg*$_c$*(open, m, close)*</u>

This step may cause the consequence of the invariant to go from true to false by assiging isn_c to a value other than i. However, when this assignment is made, $mode_c$ is also assigned the value `syn-sent`, so the invariant holds after this step.

<u>a = *receive-seg*$_{sc}$*(SYN, sn_s, ack_s)*</u>

This step may cause the premise of the invariant to go from false to true by adding (i, isn_s) to *estb-pairs* and changing $mode_c$ to be in *sync-states*. The invariant holds after this step becasue $i = isn_c$. ■

Invariant 7.35

1. If there exists $p \in \textit{in-transit}_{cs}$ such that $msg(p) \neq msg_c$ then $sn(p) < sn_c \ \vee msg_c =$ `null`.

2. If there exists $p \in \textit{in-transit}_{sc}$ such that $msg(p) \neq msg_s$ then $sn(p) < sn_s \ \vee msg_s =$ `null`.

Proof: In the start state $\textit{in-transit}_{cs}$ and $\textit{in-transit}_{sc}$ are both empty, so the invariant holds in this state. We consider critical steps of the form (s, a, s') for Part 1 below.

1. <u>a = *prepare-msg*$_c$</u>

   This step assigns msg_c to head($\textit{send-buf}_c$). However, it also increments sn_c once or twice. Therefore, $s'.sn_c > s.sn_c$. From Invariant 7.1, we know that for all $p \in$

$s.in\text{-}transit_{cs}$, $s.sn_c \geq sn(p)$. Since this step does not change any elment of $in\text{-}transit_{cs}$, we know $s'.sn_c > sn(p)$, so Part 1 holds after this step.

a = *receive-seg*$_{sc}$(sn_s, ack_s, msg_s) and a = *receive-seg*$_{sc}$(sn_s, ack_s, msg_s, *FIN*)

These steps may cause the premise of Part 1 to go from false to true because they may change msg_c to `null`. However, the consequence also clearly becomes true.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.36

1. If $msg_c \neq$ `null` and there exists $p \in in\text{-}transit_{cs}$ such that $sn(p) = sn_c$ then $msg(p) = msg_c$.

2. If $msg_s \neq$ `null` and there exists $p \in in\text{-}transit_{sc}$ such that $sn(p) = sn_s$ then $msg(p) = msg_s$.

Proof: In the start state $in\text{-}transit_{cs}$ and $in\text{-}transit_{sc}$ are both empty, so the invariant holds in this state. We consider critical steps of the form (s, a, s') for Part 1 below.

1. a = *send-seg*$_{cs}$(sn_c, ack_c, msg_c) and a = *send-seg*$_{cs}$(sn_c, ack_c, msg_c, *FIN*)

   These steps may change the premise of Part 1 from false to true by adding a segment p to $in\text{-}transit_{cs}$ with $sn(p) = sn_c$. However, for segment p, $msg(p) = msg_c$, so Part 1 holds after these steps.

   a = *receive-seg*$_{sc}$(sn_s, ack_s, msg_s) and a = *receive-seg*$_{sc}$(sn_s, ack_s, msg_s, *FIN*)

   These steps may cause the consequence of Part 1 to go from true to false because they may change msg_c to `null`. However, the premise also clearly becomes false too.

   a = *prepare-msg*$_c$

   This step may cause the consequence of Part 1 to go from true to false by assigning a new value to msg_c. However, when this assignment is made sn_c is incremented, so from Invariant 7.1 we know $sn(p) \neq sn_c$, so Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.37

1. If there exists segments p and q on $in\text{-}transit_{cs}$ such that $sn(p) = sn(q) \ \wedge msg(p) \neq$ `null` $\wedge \ msg(q) \neq$ `null` then $msg(p) = msg(q)$.

2. If there exists segments p and q on $in\text{-}transit_{sc}$ such that $sn(p) = sn(q) \ \wedge msg(p) \neq$ `null` $\wedge \ msg(q) \neq$ `null` then $msg(p) = msg(q)$.

Proof: In the initial state $in\text{-}transit_{cs}$ and $in\text{-}transit_{sc}$ are both empty, so the invariant holds for this case. We consider critical steps of the form (s, a, s') for Part 1 below.

1. $a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$

   These steps add a segment to $in\text{-}transit_{cs}$. If these steps make the premise of Part 1 go from false to true, then it must be that in $s.sn_c = sn(q)$ for some segment $q \in s.in\text{-}transit_{cs}$. From Invariant 7.35 we know that since $s.sn_c = sn(q)$, $s.msg_c = msg(q) \ \vee s.msg_c =$ `null`. Therefore, Part 1 holds after these steps.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.38

If $mode_s \in sync\text{-}states$ then $(isn_c^s, isn_s) \in assoc$.

Proof: In the start state $mode_s =$ `closed`, so the invariant holds in this state. We examine critical steps of the form (s, a, s') below.

$a = receive\text{-}seg_{cs}(sn_c{,}ack_c,\ msg_c)$ and $a = receive\text{-}seg_{cs}(sn_c{,}ack_c,\ msg_c,\ FIN)$

In these steps $mode_s$ may be assigned to an element of $sync\text{-}states$. However, if this assignment is made, (isn_c^s, isn_s) is added to $assoc$. ■

Invariant 7.39

If $mode_c =$ `syn-sent` $\wedge\ mode_s =$ `syn-rcvd` $\wedge\ isn_c \neq isn_c^s$ then for all SYN segments $p \in in\text{-}transit_{sc}$ such that $ack(p) = sn_c + 1$, $sn(p) < sn_s$.

Proof: In the start state $mode_c =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a = send\text{-}msg_c(open,\ m,\ close)$

This step can change the premise of the invariant from false to true. However, we know from Invariant 7.11 that when this happens there are no segments $p \in in\text{-}transit_{sc}$ such that

$ack(p) = sn_c + 1$. Therefore, the invariant holds after this step.

$\underline{a = \mathit{receive\text{-}seg}_{cs}(SYN,\ sn_c)}$

This is another step that can change the premise of the invariant from false to true. However from Invariant 7.1 we know that $s.sn_s \geq sn(p)$ for all $p \in s.\mathit{in\text{-}transit}_{sc}$. After this step $s.sn_s < s'.sn_s$, and since the step does not change $s.\mathit{in\text{-}transit}_{cs}$, we know $s'.sn_s > sn(p)$ for all $p \in s'.\mathit{in\text{-}transit}_{sc}$, so the invariant holds after this step.

$\underline{a = \mathit{send\text{-}seg}_{sc}(SYN,\ sn_s,\ ack_s)}$

This step can change the consequence of the invariant from true to false by adding a SYN segment to $\mathit{in\text{-}transit}_{sc}$. However, we know from Invariant 7.7 that if $mode_s$ = `syn-rcvd` then $ack_s = isn_c^s + 1$, and we know from Invariant 7.10 that if $mode_c$ = `syn-sent` then $isn_c = sn_c$. Therefore, if $ack_s = sn_c + 1$ then $isn_c = isn_c^s$, so the premise of the invariant is also false. ■

Invariant 7.40

If $mode_c \in \mathit{sync\text{-}states} \wedge mode_s =$ `syn-rcvd` $\wedge\ isn_c \neq isn_c^s$ then $ack_c < sn_s + 1$.

Proof: In the start state $mode_c$ = `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$\underline{a = \mathit{receive\text{-}seg}_{cs}(SYN,\ sn_c)}$

This step that can change the premise of the invariant from false to true. However, after this step $s'.sn_s = s.sn_s + 1$, and we know from Invariant 7.2 that $s.ack_c \leq s.sn_c + 1$. Therefore, after this step $s'.ack_c < s'.sn_s + 1$, so the invariant holds.

$\underline{a = \mathit{receive\text{-}seg}_{sc}(SYN,\ sn_s,\ ack_s)}$

This step may also change the premise of the invariant from false to true, if $[ack_s] = sn_c + 1$ and $mode_c$ = `syn-sent`. However, we know by Invariant 7.39 that $[sn_s] < sn_s$, and since this step assigns ack_c to $[sn_s] + 1$, we know the invariant holds after this step.

$\underline{a = \mathit{receive\text{-}seg}_{cs}(sn_c,\ ack_c,\ msg_c)}$ <u>or</u> $\underline{a = \mathit{receive\text{-}seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$

These steps may cause the consequence of the invariant to go from true to false, but only if $s.mode_s \neq$ `syn-rcvd`, or if $s.mode_s$ = `syn-rcvd`, the steps change $mode_s$ to `estb`, and `close-wait` respectively, so the invariant holds. ■

Invariant 7.41

If $mode_c \in$ *sync-states* $\wedge\ mode_s =$ `syn-rcvd` $\wedge\ isn_c \neq isn_c^s$ then for all segments $p \in$ *in-transit*$_{cs}$, $ack(p) < sn_s + 1$.

Proof: In the start state $mode_c =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a =$ *receive-seg*$_{cs}$*(SYN,* sn_c*)*

This step that can change the premise of the invariant from false to true. However, by Invariant 7.11 we know the consequence is also true.

$a =$ *receive-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*

This step may also change the premise of the invariant from false to true, if $[ack_s] = sn_c + 1$ and $mode_c =$ `syn-sent`. This step also assigns ack_c to $[sn_s] + 1$. From Invariant 7.23 we know that for all $p \in$ *in-transit*$_{cs}$, $ack_c \geq ack(p)$, and from Invariant 7.1 we know $sn_s \geq [sn_s]$. Therefore, after this step we know $ack(p) < sn_s + 1$.

$a =$ *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* or $a =$ *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*

These steps can change the consequence of the invariant from true to false by adding a segment p to *in-transit*$_{cs}$ with $ack(p) > sn(p)$. However, from Invariant 7.40 we know that if $mode_c \in$ *sync-states* $\wedge\ mode_s =$ `syn-rcvd` $\wedge\ isn_c \neq isn_c^s$ then $ack_c < sn_s + 1$. Therefore, if $ack_c \geq sn_s + 1$ then the premise of the invariant must be false, so the invariant holds after either of these steps. ■

Invariant 7.42

If $mode_s =$ `syn-rcvd` and there exists $p \in$ *in-transit*$_{cs}$ such that $ack(p) = sn_s + 1$, then $mode_c \neq$ `syn-sent` or for all SYN segments $q \in$ *in-transit*$_{sc}$, $ack(q) \neq sn_c + 1$.

Proof: In the start state $mode_s =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

We know the step where $mode_s$ is assigned `syn-rcvd` is not critical because Invariant 7.11 tells us that there are no segments $p \in$ *in-transit*$_{cs}$ such that $ack(p) = sn_s + 1$ when this assignment is made.

$a =$ *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* or $a =$ *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*

These steps may change the premise of the invariant from false to true. However, if these actions are enabled in state s, then we know $s.mode_c \neq$ `syn-sent`, and neither of these steps changes $mode_c$. Therefore, the invariant holds after these steps.

From Invariant 7.11 we know that the step where $mode_c$ is assigned `syn-sent` is not critical, because when that assignment is made, there are no SYN segments $q \in in\text{-}transit_{sc}$ with $ack(q) = sn_c + 1$.

<u>a = *send-seg*$_{sc}$(*SYN*, sn_s, ack_s)</u>

This step may change the consequence of the invariant from true to false, if $s.mode_s$ = `syn-rcvd` and $s.ack_s = s.sn_c + 1$. However, if these conditions hold in state s, Invariant 7.21 tells us that for all $p \in s.in\text{-}transit_{cs}$, $s.sn_s \geq ack(p)$. Therefore, after this step the premise of the invariant is also false, so the invariant holds. ■

Invariant 7.43

If $mode_c$ = `syn-sent` and there exists SYN segment $p \in in\text{-}transit_{sc}$ such that $ack(p) = sn_c + 1$, then $mode_s \notin sync\text{-}states$ and for all $i \in \mathbb{N}$ $(i, isn_s) \notin assoc$.

Proof: In the start state $mode_c$ = `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

<u>a = *send-seg*$_{sc}$(*SYN*, sn_s, ack_s)</u>

This step can make the premise of the invariant go from false to true by adding a SYN segment with $ack(p) = sn_c + 1$ to $in\text{-}transit_{sc}$. However, this step is only enabled if $mode_s$ = `syn-rcvd` and we know from Invariant 7.16 that if $mode_s$ = `syn-rcvd` then for all i, $(i, isn_s) \notin assoc$. Since this step does not add any elements to $assoc$, we know the invariant holds after this step.

<u>a = *receive-seg*$_{cs}$(sn_c, ack_c, msg_c) and a = *receive-seg*$_{cs}$(sn_c, ack_c, msg_c, *FIN*)</u>

These steps can make the consequence of the invariant go from true to false by adding (isn_c^s, isn_s) to $assoc$ and assiging $mode_s$ to a value in $sync\text{-}states$. These assigments happen if $s.mode_c$ = `syn-rcvd` and there exists a segment $p \in s.in\text{-}transit_{cs}$ with $ack(p) = s.sn_s + 1$. From Invariant 7.42 we know that if these conditions are true in state s, then either $s.mode_c \neq$ `syn-sent` or there are no SYN segment $p \in s.in\text{-}transit_{sc}$ such that $ack(p) = s.sn_c + 1$. Since these steps do not change $mode_c$ or $in\text{-}transit_{sc}$, we now the invariant holds after this step. ■

Invariant 7.44

If $mode_c \in sync\text{-}states \wedge (isn_c^s, isn_s) \in assoc$ then $isn_c = isn_c^s$.

Proof: In the start state $mode_c$ = `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. Because of Invariant 7.43, we know a = *receive-seg$_{sc}$(SYN, sn$_s$, ack$_s$)* is not a critical step.

<u>a = *receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$)* and a = *receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$, FIN)*</u>

These steps can make the premise of the invariant go from false to true by adding (isn_c^s, isn_s) to *assoc*. This assigment happens if $s.mode_c$ = `syn-rcvd` and there exists a segment $p \in s.in\text{-}transit_{cs}$ with $ack(p) = s.sn_s + 1$. From Invariant 7.41 we know that if this is the case in state s, then $s.isn_c = s.isn_c^s$. Since these steps do not change either isn_c or isn_c^s, we know the invariant holds after these steps.

<u>a = *send-msg$_c$(open, m, close)*</u>

This step can make the consequence of the invariant go from true to false, but if this happens, $mode_c$ is also assigned to `syn-sent`, so the invariant hols after this step.

<u>a = *receive-seg$_{cs}$(SYN, sn$_c$)*</u>

This can also make the consequence of the invariant go from true to false. However, from Invariant 7.16, we know that $(isn_c^s, isn_s) \notin assoc$, so the invariant holds after this step. ■

Invariant 7.45

If $mode_c$ = `syn-sent` and there exists a SYN segment $p \in in\text{-}transit_{sc}$ such that $ack(p) = sn_c + 1$ then for all non-SYN segments $q \in in\text{-}transit_{sc}$, $sn(q) < sn(p) + 1$.

Proof: In the start state $mode_c$ = `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

<u>a = *send-seg$_{sc}$(SYN, sn$_s$, ack$_s$)*</u>

This step may change the premise of the invariant from false to true by adding a SYN segment to $in\text{-}transit_{cs}$. However, from Invariant 7.1 we know that sn_s greater than or equal to $sn(q)$ for $q \in in\text{-}transit_{sc}$, so the invariant holds after this step.

<u>a = *send-seg$_{sc}$(sn$_s$, ack$_s$, msg$_s$)* and a = *send-seg$_{sc}$(sn$_s$, ack$_s$, msg$_s$, FIN)*</u>

These steps add non-SYN segments to $in\text{-}transit_{sc}$ if $s.mode_s \in sync\text{-}states$, so they can make the consequence of the invariant go from true to false. From Invariant 7.43 we know that if $s.mode_s$ is in the set of synchronized states, then the premise of the invariant is also false. ■

Invariant 7.46

If $mode_c \in$ *sync-states* and there exists a non-SYN segment $p \in$ *in-transit*$_{sc}$ such that $sn(p) \geq ack_c$ then there exists j such that $(isn_c, j) \in assoc$.

Proof: In the start state $mode_c =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We know that $a =$ *receive-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)* is not critical even though it may assign $mode_c$ to `estb`, and ack_c to $[sn_s]+1$. These assignment are made if $s.mode_c =$ `syn-sent` and $[ack_s] = sn_c + 1$. However, from Invariant 7.45 we know that in state s, all non-SYN segments $q \in s.$*in-transit*$_{sc}$, have $sn(q) < [sn_s] + 1$. Thus, the premise of the invariant does not become true after this step.

$a =$ *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and $a =$ *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)*

These steps add a non-SYN segment to *in-transit*$_{sc}$ if $s.mode_s \in$ *sync-states*, so they can make the premise of the invariant go from false to true. From Invariant 7.38 we know that if $mode_s \in$ *sync-states* then there exists i such that $(i, isn_s) \in assoc$. From Invariant 7.31 we know that if $(i, isn_s) \in assoc$ then $i = isn_c^s$, and from Invariant 7.44 we know that $isn_c = isn_c^s$. Therefore, the invariant holds after these steps. ■

Invariant 7.47

If $mode_c \in \{$`close-wait`, `closing`, `last-ack`, `timed-wait`$\}$ then $\exists\ j$ such that $(isn_c, j) \in assoc$.

Proof: In the start state $mode_c =$ `closed`, so the invariant holds in this state. We examine critical steps of the form (s, a, s') below.

$a =$ *receive-seg*$_{sc}$*(*sn_s*,*ack_s*,* msg_s*, FIN)*

This step can make the premise of the invariant go from false to true. However, from Invariant 7.46, we know that the consequence is also true, so the invariant holds. ■

Invariant 7.48

If $mode_s =$ `syn-rcvd` $\wedge\ isn_c = isn_c^s\ \wedge\ ack_c = isn_s + 1$ then $mode_c \notin \{$`close-wait`, `closing`, `last-ack`, `timed-wait`$\}$.

Proof: In the start state $mode_s$ has the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. The $a =$ *receive-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* step is not critical even though it may cause ack_c to be assigned $[sn_s] + 1$. We know it is not critical because Invariant 7.25 tells us that if this assignment is made, then either $mode_s \neq$ `syn-rcvd` or $ack_c < isn_s + 1$.

<u>a = *receive-seg*$_{sc}$(*SYN*, sn_s, ack_s)</u>

This step may assign ack_c the value $sn_s + 1$. However, $mode_c$ is also assigned `estb`, so the invariant holds after this step.

<u>a = *receive-seg*$_{sc}$(sn_s, ack_s, msg_s, *FIN*)</u>

This step may cause the consequence of the invariant to go from true to false. This change happens if $s.mode_c \in \{$`estb`, `fin-wait-1`, `fin-wait-2`$\}$ and $[sn_s] \geq ack_c$. However, form Invariant 7.25 we know that if these conditions are true in state s, then either $mode_s \neq$ `syn-rcvd` or $ack_c < isn_s + 1$. ■

Invariant 7.49

If $mode_s$ = `syn-rcvd` $\wedge$ $isn_c = isn_c^s$ and there exists a non-SYN segment $p \in in\text{-}transit_{cs}$ such that $ack(p) = sn_s + 1$ then $mode_c \notin \{$`close-wait`, `closing`, `last-ack`, `timed-wait`$\}$.

Proof: In the start state $mode_s$ has the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

<u>a = *send-seg*$_{cs}$(sn_c, ack_c, msg_c) and a = *send-seg*$_{cs}$(sn_c, ack_c, msg_c, *FIN*)</u>

These steps may cause the premise of the invariant to go from false to true. However, we know from Invariant 7.48 that the consequence is also true after these steps.

<u>a = *receive-seg*$_{sc}$(sn_s, ack_s, msg_s, *FIN*)</u>

This step may cause the consequence of the invariant to go from true to false. This change happens if $s.mode_c \in \{$`estb`, `fin-wait-1`, `fin-wait-2`$\}$ and $[sn_s] \geq ack_c$. However, form Invariant 7.25 we know that if these conditions are true in state s, then either $mode_s \neq$ `syn-rcvd` or $ack_c < isn_s+1$, and from Invariant 7.23 we know $ack_c \geq ack(p)$ for any segment $p \in in\text{-}transit_{cs}$. Therefore, the invariant holds after this step. ■

Invariant 7.50

If $mode_s$ = `syn-rcvd` $\wedge$ $mode_c \notin \{$`closed`, `rec`, `reset`$\}$ and there exists a non-SYN segment $p \in in\text{-}transit_{cs}$ such that $ack(p) = sn_s + 1$ and there exists a FIN segment $q \in in\text{-}transit_{cs}$ such that $(sn(q) \geq \max(ack_s, sn(p) + 1) \vee (p = q \wedge sn(q) \geq ack_s))$ then $mode_c \in$

{$\texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}$}.

Proof: In the start state $mode_c$ is `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We know a = *receive-msg*$_{cs}$*(SYN, sn*$_c$*)* where $mode_s$ is assigned `syn-rcvd` is not critical because Invariant 7.11 tells us that after this step there are no segments $p \in$ *in-transit*$_{cs}$ such that $ack(p) = sn_s + 1$. We also know that a = *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* is not critical because Invariant 7.1 tells us that there cannot be a segment $q \in$ *in-transit*$_{cs}$ with $sn(q) > [sn_c]$, which must be the case if this step makes the premise of the invariant true.

a = *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)*

This step adds a FIN segment to *in-transit*$_{cs}$, but only if $mode_c \in$ {`fin-wait-1`, `closing`, `last-ack`}, so the invariant holds after this step.

Steps that cause $mode_s$ to be in the set {`closed`, `rec`, `reset`} are also critical because the cause the consequence of Part 1 to go from true to false, but the premise also clearly becomes false after any of these steps, so the invariant still holds.

Invariant 7.51

1. If $mode_c \in$ *sync-states* $\wedge$ $mode_s \notin$ {`rec`, `reset`} $\wedge$ $(isn_c, isn_s) \in$ *assoc* and there exists a FIN segment $p \in$ *in-transit*$_{sc}$ such that $sn(p) \geq ack_c$ then $mode_s \in$ {`fin-wait-1`, `fin-wait-2`, `closing`, `timed-wait`, `last-ack`}.

2. If $mode_s \in$ *sync-states* $\wedge$ $mode_c \notin$ {`rec`, `reset`} $\wedge$ $(isn_c, isn_s) \in$ *estb-pairs* $\wedge$ $isn_c = isn_c^s$ and there exists a FIN segment $p \in$ *in-transit*$_{cs}$ such that $sn(p) \geq ack_s$ then $mode_c \in$ {`fin-wait-1`, `fin-wait-2`, `closing`, `timed-wait`, `last-ack`}.

Proof: In the start state $mode_c$ and $mode_s$ are `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We know that a = *receive-seg*$_{sc}$*(SYN, sn*$_s$*, ack*$_s$*)* is not critical even though it may assign $mode_c$ to `estb`, and ack_c to $[sn_s] + 1$. These assignment are made if $s.mode_c$ = `syn-sent`. From Invariant 7.16 we know that if $s.mode_c$ = `syn-sent`, then $(isn_c, isn_s) \notin$ *assoc*. Thus, the premise of the invariant does not become true after this step. Steps with a = *receive-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* and a = *receive-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*, FIN)* may cause (isn_c, isn_s) to be added to *assoc* if $[ack_c] > sn_s$. However, these steps do not make the premise of Part 1 go from false to true, because from Invariant 7.23 we know that $ack_c \geq [ack_c]$, and from Invariant 7.1 we know $sn_s \geq sn(p)$ for all $p \in$ *in-transit*$_{sc}$. Therefore, if these steps cause (isn_c, isn_s) to be added to *assoc* then we know there are no segments $p \in$ *in-transit*$_{sc}$ such that $sn(p) \geq ack_c$.

1. $a =$ *send-seg*$_{sc}$*(*sn_s, ack_s, msg_s, *FIN)*

   This step adds a FIN segment to *in-transit*$_{sc}$, but only if $s.mode_s \in$ {`fin-wait-1`, `last-ack`, `closing`}, so Part 1 obviously holds after this step.

   Steps that cause $mode_s$ to be in the set {`closed`, `rec`, `reset`} are also critical because the cause the consequence of Part 1 to go from true to false, but the premise also clearly becomes false after any of these steps, so Part 1 still holds.

2. For Part 2 we know that $a =$ *receive-seg*$_{sc}$*(SYN,* sn_s, ack_s*)* is not critical even though this step may cause (isn_c, isn_s) to be in *estb-pairs* if $mode_c =$ `syn-rcvd`. However, we know from Invariant 7.19 that if this is the case in state s, then $mode_s \notin$ *sync-states*, so the premise does not become true after this step.

   $a =$ *send-seg*$_{cs}$*(*sn_c, ack_c, msg_c, *FIN)*

   Symmetric to the case for $a =$ *send-seg*$_{sc}$*(*sn_s, ack_s, msg_s, *FIN)* of Part 1.

   $a =$ *receive-seg*$_{cs}$*(*sn_c, ack_c, msg_c*)* and $a =$ *receive-seg*$_{cs}$*(*sn_c, ack_c, msg_c, *FIN)*

   These steps may cause $mode_s$ to be in *sync-states* if $s.mode_s =$ `syn-rcvd`. However, we know from Invariant 7.50, that if this is the case, then the consequence of Part 2 is also true.

   Steps that cause $mode_c$ to be in the set {`closed`, `rec`, `reset`} are also critical because the cause the consequence of Part 2 to go from true to false, but the premise also clearly becomes false after any of these steps, so Part 2 still holds. ■

Invariant 7.52

1. If $mode_c \in$ {`close-wait`, `closing`, `last-ack`, `timed-wait`} $\wedge$ $mode_s \notin$ {`rec`, `reset`} $\wedge$ $(isn_c, isn_s) \in$ *assoc* then $mode_s \in$ {`fin-wait-1`, `fin-wait-2`, `closing`, `timed-wait`, `last-ack`}.

2. If $mode_s \in$ {`close-wait`, `closing`, `last-ack`, `timed-wait`} $\wedge$ $mode_c \notin$ {`rec`, `reset`} $\wedge$ $(isn_c, isn_s) \in$ *estb-pairs* $\wedge$ $isn_c = isn_c^s$ then $mode_c \in$ {`fin-wait-1`, `fin-wait-2`, `closing`, `timed-wait`, `last-ack`}.

Proof: In the initial state $mode_s$ and mode$_c$ are `closed`, so the invariant holds. We consider critical actions of the form (s, a, s') below. For Part 1 we know that a = *receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$)* and a = *receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$, FIN)* are not critical even though they may cause the pair (isn_c, isn_s) to be added to *assoc*. The pair is added if $s.mode_s$ = `syn-rcvd` $\wedge\ [ack_c] = s.sn_s + 1$. However, if these conditions are true in state s, then Invariant 7.49 tells us that $s.mode_c \notin$ {`close-wait`, `closing`, `last-ack`, `timed-wait`}, and since these steps do not change $mode_c$, we know the premise does not become true.

1. <u>a = *receive-seg$_{sc}$(sn$_s$, ack$_s$, msg$_s$, FIN)*</u>

   This step may cause the premise of the invariant to go from false to true. This change happens if $s.mode_c \in$ *sync-states* and $[sn_s] \geq s.ack_c$. Therefore, from Invariant 7.51 we know that $s.mode_s \in$ {`fin-wait-1`, `fin-wait-2`, `closing`, `timed-wait`, `last-ack`}. Since step does not change $mode_s$, we know Part 1 holds after this step.

   Steps that cause $mode_s$ to be in the set {`closed`, `rec`, `reset`} are also critical because the cause the consequence of Part 1 to go from true to false, but the premise also clearly becomes false after any of these steps, so Part 1 still holds.

2. For Part 2 we know that a = *receive-seg$_{sc}$(SYN, sn$_s$, ack$_s$)* is not critical even though this step may cause (isn_c, isn_s) to be in *estb-pairs* if $mode_c$ = `syn-rcvd`. However, we know from Invariant 7.19 that if this is the case in state s, then $mode_s \notin$ *sync-states*, so the premise does not become true after this step.

   <u>a = *receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$, FIN)*</u>

   The proof that Part 2 holds after this step is symmetric to the proof that Part 1 holds after a = *receive-seg$_{sc}$(sn$_s$, ack$_s$, msg$_s$, FIN)*.

   Steps that cause $mode_c$ to be in the set {`closed`, `rec`, `reset`} are also critical because the cause the consequence of Part 2 to go from true to false, but the premise also clearly becomes false after any of these steps, so Part 2 still holds. ■

Invariant 7.53

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup \mathit{sync\text{-}states} \wedge mode_c \in \{\texttt{rec}, \texttt{reset}\} \cup \mathit{sync\text{-}states} \wedge isn_c = isn_c^s \wedge isn_s = isn_s^c$ and there exists $p \in \mathit{in\text{-}transit}_{cs}$ such that $sn(p) \geq ack_s$, then $sn_c = sn(p)$.

2. If $mode_c \in \mathit{sync\text{-}states} \wedge (isn_c, isn_s) \in assoc$ and there exists $p \in \mathit{in\text{-}transit}_{sc}$ such that $sn(p) \geq ack_c$, then $sn_s = sn(p)$.

Proof: In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') for Part 1 below. The step with a = *receive-seg*$_{cs}$*(SYN,* sn_c*)* is not critical for Part 1 because after this step $isn_s \neq isn_s^c$ (Invariant 7.9). The step with a = *receive-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)* is also not critical. This step causes $mode_c$ to be in *sync-states* and isn_s^c to be isn_s only if $s.mode_c$ = `syn-sent` and $[ack_s] = s.sn_c + 1$. From Invariant 7.1 we know $sn_c \geq sn(p)$ for all $p \in \mathit{in\text{-}transit}_{cs}$, and from Invariant 7.10 we know that $s.isn_c = s.sn_c$. Finally, form Invariant 7.6 we know $ack_s > isn_c^s$. Since $isn_c^s = isn_c$, $s.isn_c \geq sn(p)$ for any $p \in s.\mathit{in\text{-}transit}_{cs}$, and $ack_s > isn_c^s$, we know that after this step there are no segments $p \in \mathit{in\text{-}transit}_{cs}$ such that $sn(p) \geq ack_s$. The steps where $a = crash_c$ or a = *receive-seg*$_{sc}$*(RST,* ack_s*,* rst-seq$_s$*)* make change $mode_c$ from `syn-sent` to `rec`, or `reset` respectively. However, these steps are not critical, because if $s.mode_c$ = `syn-sent` then $s.isn_s^c$ is equal to `nil`, but since $mode_s \in \{\texttt{syn-rcvd}\} \cup \mathit{sync\text{-}states}$, we know $isn_s \neq$ `nil` (Invariant 7.5). Therefore, these steps do not make the premise go from false to true.

1. <u>a = *send-msg*$_c$*(open, m, close)*</u>

   This step may change the consequence of the invariant from true to false by incrementing sn_c. However, if this change happens in this step, then $mode_c$ is assigned `syn-sent`, so the invariant holds after this step.

   <u>a = *prepare-msg*$_c$</u>

   This step is enabled if $s.mode_c \in \{\texttt{estb}, \texttt{close-wait}\} \wedge \neg s.\mathit{ready\text{-}to\text{-}send}_c$. We know from Invariant 7.30 that if this condition is true in state s, then $s.sn_c < s.ack_s$. From Invariant 7.1 we know $s.sn_c \geq sn(p)$ for any $p \in s.\mathit{in\text{-}transit}_{cs}$. Therefore, the premise of the invariant, is false before and after this step.

   <u>a = *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and a = *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)*</u>

These steps can make the premise of the invariant go from false to true, by adding a segment p to $in\text{-}transit_{cs}$ such that $sn(p) \geq ack_s$. However, $sn_c = sn(p)$, so the invariant holds after these steps.

2. Before we examine the critical actions for Part 2, we point out that $a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$ is not critical because from Invariant 7.16 we know that $(isn_c, isn_s) \notin assoc$ before and after this step. We also know that $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$ are not critical even though these steps can cause (isn_c, isn_s) to be added to $assoc$ if $[ack_c] = sn_s + 1$. We know these steps are not critical because from Invariant 7.29 we know that $ack_c \geq [ack_c]$ and from Invariant 7.1 we know that for any segment $p \in in\text{-}transit_{sc}$, $sn_s \geq sn(p)$. Therfore, after these steps we know there are no segments $p \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$.

   $\underline{a = receive\text{-}seg_{cs}(SYN,\ sn_c)}$

   This step may change the consequence of Part 2 from true to false, but from Invariant 7.16 we know that $(isn_c, isn_s) \notin assoc$, so the premise is also false after this step.

   $\underline{a = prepare\text{-}msg_s}$

   The proof that the Part 2 holds after this step is symmetric to the proof for $a = prepare\text{-}msg_c$ for Part 1.

   $\underline{a = send\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)}$ and $\underline{a = send\text{-}seg_{cs}(sn_s,\ ack_s,\ msg_s)}$

   These steps can make the premise of the invariant go from false to true, by adding a segment p to $in\text{-}transit_{cs}$ such that $sn(p) \geq ack_s$. However, $sn_s = sn(p)$, so Part 2 holds after these steps. ■

Invariant 7.54

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states \ \wedge mode_c \in sync\text{-}states \ \wedge \ (ready\text{-}to\text{-}send_c \vee send\text{-}fin_c) \wedge \ (isn_c, isn_s) \in estb\text{-}pairs \ \wedge isn_c^s = isn_c \ \wedge \ ((sn_c = ack_s \wedge \neg(rcvd\text{-}close_c \wedge send\text{-}buf_c = \epsilon)) \vee sn_c = ack_s + 1)$ then $msg_c \neq \texttt{null}$.

2. If $mode_c \in sync\text{-}states \wedge (isn_c, isn_s) \in assoc \ \wedge \ (ready\text{-}to\text{-}send_s \vee send\text{-}fin_s) \wedge \ ((sn_s = ack_c) \wedge \neg(rcvd\text{-}close_s \wedge send\text{-}buf_s = \epsilon)) \vee (sn_s = ack_c + 1)$ then $msg_s \neq \texttt{null}$.

Proof: In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') for Part 1 below.

1. $\underline{a = \textit{prepare-msg}_c}$

   This step may cause the premise of the invariant to go from false to true, but msg_c also gets assigned to head($\textit{send-buf}_c$), so it is not equal to `null`. Therefore, Part 1 holds after this step.

   $\underline{a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s) \text{ and } a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)}$

   These steps may cause the consequence of Part 1 to go from true to false. However, they also make the premise false, so Part 1 holds after these steps.

2. The proof for Part 2 is symmetric. ∎

Invariant 7.55

1. If $mode_s \in \textit{sync-states}$ and there exists non-FIN segment $p \in \textit{in-transit}_{cs}$ such that $sn(p) = ack_s$ or a FIN segment $p \in \textit{in-transit}_{cs}$ such that $sn(p) = ack_s + 1$ then $msg(p) \neq \texttt{null}$.

2. If $mode_c \in \textit{sync-states}$ and there exists non-FIN segment $p \in \textit{in-transit}_{sc}$ such that $sn(p) = ack_c$ or a FIN segment $p \in \textit{in-transit}_{sc}$ such that $sn(p) = ack_c + 1$ then $msg(p) \neq \texttt{null}$.

Proof: In the start state $mode_c = mode_s = \texttt{closed}$, so the invariant holds in this state. We examine critical steps of the form (s, a, s') below.

1. $\underline{a = \textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c) \text{ and } a = \textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$

   These can make the premise of the Part 1 go from false to true by adding a segment to $\textit{in-transit}_{cs}$ that satisfies the properties of Part 1. From Invariant 7.54 we know that if, the consequence of Part 1 is also true.

2. The proof for Part 2 is symmetric. ∎

Invariant 7.56

If $mode_s$ = $\texttt{syn-rcvd}$ and there exists non-FIN segment $p \in$ *in-transit*$_{cs}$ such that $sn(p) = ack_s$ or a FIN segment $p \in$ *in-transit*$_{cs}$ such that $sn(p) = ack_s + 1$ and $ack(p) = sn_s + 1$ then $msg(p) \neq \texttt{null}$.

Proof: The proof is the same as the proof of Part 1 of Invariant 7.55. ■

Invariant 7.57

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ $\wedge\ (isn_c, isn_s) \in assoc$ then $sn_s < ack_c$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\}$ $\wedge\ (isn_c, isn_s) \in$ *estb-pairs* $\wedge\ isn_c = isn_c^s$ then $sn_c < ack_s$.

Proof: In the initial state $mode_s$ and mode$_c$ are $\texttt{closed}$, so the invariant holds. We consider critical actions of the form (s, a, s') below. For Part 1 we know that $a =$ *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c)* and $a =$ *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c, FIN)* are not critical even though they may cause the pair (isn_c, isn_s) to be added to *assoc*. The pair is added if $s.mode_s = \texttt{syn-rcvd} \wedge [ack_c] = s.sn_s + 1$. However, if these conditions are true in state s, then Invariant 7.49 tell us that $s.mode_c \notin \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$, and since these steps do not change $mode_c$, we know the premise does not become true.

1. <u>$a =$ *receive-seg*$_{cs}$*(SYN, sn_c)*</u>

   This step may cause the consequence of Part 1 to go from true to false by assigning sn_s to a value greater than ack_c. However, we from Invariant 7.16 that when this assignment happens, $(isn_c, isn_s) \notin assoc$, so the invariant holds after this step.

   <u>$a =$ *receive-seg*$_{sc}$*(SYN, sn_s, ack_s)*</u>

   This step may cause the consequence of Part 1 to go from true to false by assigning ack_c a value less than or equal to sn_s. However, we know from Invariant 7.16 that when this assignment happens, $(isn_c, isn_s) \notin assoc$, so the invariant holds after this step.

   <u>$a =$ *receive-seg*$_{sc}$*(sn_s, ack_s, msg_s, FIN)*</u>

   This step may cause the premise of Part 1 to go from false to true if $[sn_s] \geq ack_c$. From

Invariant 7.53 we know that $sn_s = [sn_s]$, and this step also assigns ack_c to $[sn_s] + 1$, therefore Part 1 holds after this step.

$\underline{a = \textit{prepare-msg}_s}$

This step may also cause the consequence of Part 1 to go from true to false by assigning sn_s to a value greater than ack_c. This step is enabled if $mode_s \in \{\texttt{estb}, \texttt{close-wait}\}$. However, from Invariant 7.52 we know that if $mode_s$ is in this set, then the premise of Part 1 is also false.

2. We consider the critical actions for Part 2 below. The proof is mostly symmetric to the proof for Part 1. We know that $a = \textit{receive-seg}_{sc}(SYN,\ sn_s,\ ack_s)$ is not critical even though this step may cause (isn_c, isn_s) to be in *estb-pairs* if $mode_c = \texttt{syn-rcvd}$. However, we know from Invariant 7.19 that if this is the case in state s, then $mode_s \notin$ *sync-states*, so the premise does not become true after this step.

   $\underline{a = \textit{send-msg}_c(open,\ m,\ close)}$

   This step may cause the consequence of Part 2 to go from true to false by assigning sn_s to a value greater than ack_c. However, we from Invariant 7.16 that when this assignment happens, $(isn_c, isn_s) \notin$ *estb-pairs*, so the invariant holds after this step.

   $\underline{a = \textit{receive-seg}_{cs}(SYN,\ sn_c)}$

   This step may cause the consequence of Part 2 to go from true to false by assigning ack_s a value less than or equal to sn_c. However, we know from Invariant 7.16 that when this assignment happens, $(isn_c, isn_s) \notin$ *estb-pairs*, so the invariant holds after this step.

   $\underline{a = \textit{receive-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$

   The proof for this case is symmetric to the case for $a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$ of Part 1.

   $\underline{a = \textit{prepare-msg}_c}$

   The proof for this case is symmetric to the case for $a = \textit{prepare-msg}_s$ of Part 1. ■

Invariant 7.58

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{timed-wait}\} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in$ $assoc$ then $push\text{-}data_c = \texttt{true} \vee rcv\text{-}buf_c = \epsilon$.

2. If If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{timed-wait}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in$ $estb\text{-}pairs \wedge isn_c = isn_c^s$ then $push\text{-}data_s = \texttt{true} \vee rcv\text{-}buf_s = \epsilon$.

Proof: In the initial state $mode_s$ and mode$_c$ are `closed`, so the invariant holds. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)}$

   This step may cause the consequence of Part 1 to go from true to false, but only if $[sn_s] = ack_c$. We know from Invariant 7.1 that $[sn_s] \leq sn_s$, and we know from Invariant 7.57 that if the premise of Part 1 is true, then $sn_s < ack_c$. Therefore, if $[sn_s] = ack_c$, then the premise must be false, so Part 1 holds after this step.

   $\underline{a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)}$

   This step may cause the premise of Part 1 to go from false to true. However, this step also sets $push\text{-}data_c$ to `true`. Therefore, Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.59

1. If $mode_c = \texttt{last-ack} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in assoc$ then $rcv\text{-}buf_c = \epsilon$.

2. If $mode_s = \texttt{last-ack} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (isn_c, isn_s) \in estb\text{-}pairs \wedge isn_c = isn_c^s$ then $rcv\text{-}buf_s = \epsilon$.

Proof: In the initial state $mode_s$ and mode$_c$ are `closed`, so the invariant holds. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = prepare\text{-}msg_c}$

   This step may change the premise of Part 1 from false to true. This change happens if $s.mode_c = \texttt{close-wait} \wedge \neg s.push\text{-}data_c$. However, if these conditions are true in state s, then from Invariant 7.58, we know that $rcv\text{-}buf_c = \epsilon$, so Part 1 holds after this step.

$\underline{a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s) \text{ and } a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)}$

These steps may cause the consequence of Part 1 to go from true to false, but only if $[sn_s] \geq ack_c$. We know from Invariant 7.1 that $[sn_s] \leq sn_s$, and we know from Invariant 7.57 that if the premise of Part 1 is true, then $sn_s < ack_c$. Therefore, if $[sn_s] \geq ack_c$, then the premise must be false, so Part 1 holds after these steps.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.60

1. If $mode_c = \texttt{closing} \wedge send\text{-}fin\text{-}ack_c = \texttt{true}$ then $rcv\text{-}buf_c = \epsilon$.

2. If $mode_s = \texttt{closing} \wedge send\text{-}fin\text{-}ack_c = \texttt{true}$ then $rcv\text{-}buf_s = \epsilon$.

Proof: In the initial state $mode_s$ and mode$_c$ are `closed`, so the invariant holds. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)}$

   These steps may cause the premise of Part 1 to go from false to true by assigning $send\text{-}fin\text{-}ack_c$ to `true`. This assigment is made if $\neg push\text{-}data_c$. Therefore, by Invariant 7.58 we know that $rcv\text{-}buf_c = \epsilon$, so Part 1 holds after this step.

   $\underline{a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s) \text{ and } a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)}$

   These steps may cause the consequence of Part 1 to go from true to false, but only if $[sn_s] \geq ack_c$. We know from Invariant 7.1 that $[sn_s] \leq sn_s$, and we know from Invariant 7.57 that if the premise of Part 1 is true, then $sn_s < ack_c$. Therefore, if $[sn_s] \geq ack_c$, then the premise must be false, so Part 1 holds after these steps.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.61

1. If $mode_c = \texttt{timed-wait} \wedge first(t\text{-}out_c) \in \texttt{T}$ then $rcv\text{-}buf_c = \epsilon$.

2. If $mode_s = \texttt{timed-wait} \wedge first(t\text{-}out_s) \in \texttt{T}$ then $rcv\text{-}buf_s = \epsilon$.

Proof: In the initial state $mode_s$ and mode$_c$ are `closed`, so the invariant holds. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = \textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c)}$

   This step may cause the premise of the Part 1 to go from false to true by assigning *first(t-out$_c$)* to $now_c + 2\mu$. This assignment happens if $mode_c =$ `timed-wait` $\wedge$ $\neg$*push-data*$_c$. From Invariant 7.58 we know that if these conditions are true, then *rcv-buf*$_c = \epsilon$.

   $\underline{a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s)}$

   This step may cause the premise of the Part 1 to go from false to true by assigning *first(t-out$_c$)* to now_c. This assignment happens if $mode_c =$ `closing` $\wedge$ *send-fin-ack*$_c =$ `true`. From Invariant 7.60 we know that if these conditions are true, then *rcv-buf*$_c = \epsilon$. This step may also cause the consequence of Part 1 to go from true to false, but only if $[sn_s] = ack_c$. We know from Invariant 7.1 that if the premise of Part 1 is true, then $sn_s < ack_c$. Therefore, if $[sn_s] = ack_c$, then the premise must be false, so Part 1 holds after this step.

   $\underline{a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)}$

   This step may also cause the consequence of Part 1 to go from true to false, but only if $[sn_s] = ack_c$. We know from Invariant 7.1 that if the premise of Part 1 is true, then $sn_s < ack_c$. Therefore, if $[sn_s] = ack_c$, then the premise must be false, so Part 1 holds after this step.

2. The proof for Part 2 is symmetric. ∎

Invariant 7.62

1. If $mode_c \in$ *sync-states* and there exists j such that $(isn_c, j) \in$ *assoc* and there exists a non-SYN segment $p \in$ *in-transit*$_{sc}$ such that $sn(p) \geq ack_c$, then for all non-SYN segments $q \in$ *in-transit*$_{sc}$ $sn(q) \leq sn(p)$.

2. If $mode_s \in \{$`syn-rcvd`$\} \cup$ *sync-states* and there exists i, such that $i = isn_c^s \wedge (i, isn_s) \in$ *estb-pairs* and there exists a non-SYN segment $p \in$ *in-transit*$_{cs}$ such that $sn(p) \geq ack_s$, then for all non-SYN segments $q \in$ *in-transit*$_{cs}$ $sn(q) \leq sn(p)$.

Proof: In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s')

1. a = *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c)* and a = *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c, FIN)*

   These steps may cause the premise of the invariant to go from false to true by adding (isn_c, isn_s) to *assoc*. If the pair is added to *assoc*, then these steps also cause $mode_s$ to be in a synchronized state. From Invariant 7.53 we know that if $mode_s \in$ *sync-states* and the rest of the premise of Part 1 is true, then we know that $sn_s = sn(p)$. Thus, from Invariant 7.1 we know that the consequence of Part 1 is also true.

   a = *send-seg*$_{sc}$*(sn_s, ack_s, msg_s)* and a = *send-seg*$_{sc}$*(sn_s, ack_s, msg_s,FIN)*

   These steps may cause the premise of Part 1 to go from false to true, or the consequence of Part 1 to go from true to false, but these steps are only enabled if $mode_s \in$ *sync-states*. If $mode_s \in$ *sync-states*, then by Invariants 7.38 and 7.44 we know $(isn_c, isn_s) \in$ *assoc*. Thus, by Invariants 7.53 and 7.1 we know Part 1 remains true after these steps.

2. a = *receive-seg*$_{sc}$*(SYN, sn_s, ack_s, msg_s)*

   This step may cause (i, isn_s) to be added to *estb-pairs*. This step also changes $mode_c$ to `estb`. From Invariant 7.53 we know that if $mode$ = `estb` and the rest of the premise of Part 2 is true, then we know that $sn_c = sn(p)$. Thus, from Invariant 7.1 we know that the consequence of Part 2 is also true.

   a = *send-seg*$_{cs}$*(sn_c, ack_c, msg_c)* and a = *send-seg*$_{cs}$*(sn_c, ack_c, msg_c,FIN)*

   These steps may cause the premise of Part 2 to go from false to true, or the consequence of Part 2 to go from true to false, but these steps are only enabled if $mode_c \in$ *sync-states*. From Invariant 7.34 we know that if $mode_c \in$ *sync-states* and the premise of the Part 2 is true, then $(isn_c, isn_s) \in$ *estb-pairs*. Therefore, from Invariants 7.53 and 7.1 we know Part 2 remains true after these steps. ■

Invariant 7.63

1. If $mode_c \in \{\texttt{rec}, \texttt{reset}\} \cup$ *sync-states* $\wedge$ $mode_s \in \{\texttt{syn-rcvd}\} \cup$ *sync-states* $\wedge$ $isn_c = isn_c^s \wedge (isn_c, isn_s) \in$ *estb-pairs* $\wedge$ $sn_c = ack_s + 1$ then for all non-SYN segments $p \in$

$in\text{-}transit_{cs}$, $sn(p) \neq ack_s$.

2. If $mode_c \in sync\text{-}states \wedge (isn_c, isn_s) \in assoc \wedge sn_s = ack_c + 1$ then for all non-SYN segments $p \in in\text{-}transit_{sc}$, $sn(p) \neq ack_c$.

Proof: In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') for Part 1 below. The step with $a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$ is not critical for Part 1 because after this step $sn_c \neq ack_s + 1$. The steps with $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$ are also not critical, even though they may cause ack_s to be incremented. We know they are not critical because, they cause ack_s to be incremented if $[sn_c] = ack_s$, and we know from Invariant 7.53 that if $[sn_c] = ack_s$ then $sn_c = [sn_c]$, so after these steps $sn_c \neq ack_s + 1$.

1. $\underline{a = prepare\text{-}msg_c}$

   This step may make the premise of Part 1 go from true to false. The step is enabled if $s.mode_c \in \{\texttt{estb}, \texttt{close-wait}\} \wedge \neg s.ready\text{-}to\text{-}send_c$. We know from Invariant 7.30 that if these condition are true in state s, then $s.sn_c < s.ack_s$. From Invariant 7.1 we know $s.sn_c \geq sn(p)$ for any $p \in s.in\text{-}transit_{cs}$. Since the step does not add any elements to $in\text{-}transit_{cs}$, we know the consequence is also true after this step.

   $\underline{a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c) \text{ and } a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c, FIN)}$

   These steps may change the premise of Part 1 from false to true by adding a non-SYN segment p with $sn(p) = ack_c$. However, the premise of the invariant is also clearly false if this is the case.

2. The step with $a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$ is not critical for Part 2 even though it may cause $mode_c$ to become `estb`, if $s.mode_c = \texttt{syn-sent}$. It is not critical because Invariant 7.16 tells us that $(isn_c, isn_s) \notin assoc$ in states s, and since the step does not add any elements to $assoc$, we know $(isn_c, isn_s) \notin assoc$ after the step. The steps $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c, FIN)$ may cause (isn_c, isn_s) to be added to $assoc$, if $[ack_c] = sn_s + 1$. Since Invariant 7.23 tells us that $ack_c \geq [ack_c]$ we know that after this step $sn_s \neq ack_c + 1$. The steps $a = receive$-

seg_{sc}(sn_s, ack_s, msg_s) and a = *receive-seg_{sc}(sn_s, ack_s, msg_s,FIN)* may cause ack_s to be incremented but Invariant 7.53 tells us that when this happens $sn_s \neq ack_c + 1$.

a = *prepare-msg_s*

This case is symmetric to the case for a = *prepare-msg_c* for Part 1.

a = *send-seg_{sc}(sn_s, ack_s, msg_s)* and a = *send-seg_{sc}(sn_s, ack_s, msg_s,FIN)*

These cases are symmetric to the cases with the symmetric actions for Part 1. ∎

Invariant 7.64

1. If $mode_c \in$ *sync-states* and there exists j such that $(isn_c, j) \in$ *assoc* and there exists a non-SYN segment $p \in$ *in-transit$_{sc}$* such that $sn(p) = ack_c + 1$, then for all non-SYN segments $q \in$ *in-transit$_{sc}$*, $sn(q) \neq ack_c$.

2. If $mode_s \in \{$`syn-rcvd`$\} \cup$ *sync-states* and there exists i, such that $i = isn_c^s \wedge (i, isn_s) \in$ *estb-pairs* and there exists a non-SYN segment $p \in$ *in-transit$_{cs}$* such that $sn(p) = ack_s + 1$, then for all non-SYN segments $q \in$ *in-transit$_{cs}$*, $sn(q) \neq ack_s$.

Proof: In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') for Part 1 below.

1. a = *receive-seg_{cs}(sn_c, ack_c, msg_c)* and a = *receive-seg_{cs}(sn_c, ack_c, msg_c, FIN)*

   These steps may cause the premise of the invariant to go from false to true by adding (isn_c, isn_s) to *assoc*. If the pair is added to *assoc*, then these steps also cause $mode_s$ to be in a synchronized state. From Invariant 7.53 we know that if $mode_s \in$ *sync-states* and the rest of the premise of Part 1 is true, then we know that $sn_s = sn(p)$, and from Invariant 7.63 we know that if $sn_s = ack_c + 1$ then for all non-SYN segments $q \in$ *in-transit$_{sc}$*, $sn(q) \neq ack_c$. Thus, Part 1 holds after these steps.

   a = *send-seg_{sc}(sn_s, ack_s, msg_s)* and a = *send-seg_{sc}(sn_s, ack_s, msg_s,FIN)*

   These steps may cause the premise of Part 1 to go from false to true, or the consequence of Part 1 to go from true to false, but these steps are only enabled if $mode_s \in$ *sync-states*. If $mode_s \in$ *sync-states*, then by Invariants 7.38 and 7.44 we know $(isn_c, isn_s) \in$ *assoc*. Thus, by Invariant 7.63 we know Part 1 remains true after these steps.

2. a = *receive-seg*$_{sc}$*(SYN, sn_s, ack_s, msg_s)*

   This step may cause (isn_c, isn_s) to be added to *estb-pairs*. This step also changes $mode_c$ to `estb`. From Invariant 7.53 we know that if $mode$ = `estb` and the rest of the premise of Part 2 is true, then we know that $sn_c = sn(p)$. Thus, from Invariant 7.63 we know that the consequence of Part 2 is also true.

   a = *send-seg*$_{cs}$*(sn_c, ack_c, msg_c)* and a = *send-seg*$_{cs}$*(sn_c, ack_c, msg_c,FIN)*

   These steps may cause the premise of Part 2 to go from false to true, or the consequence of Part 2 to go from true to false, but these steps are only enabled if $mode_c \in$ *sync-states*. From Invariant 7.34 we know that if $mode_c \in$ *sync-states* and the premise of the Part 2 is true, then $(isn_c, isn_s) \in$ *estb-pairs*. Therefore, from Invariant 7.63 we know Part 2 remains true after these steps. ∎

Invariant 7.65

1. If $mode_c \in$ {`close-wait`, `closing`, `last-ack`, `timed-wait`} and there exists j such that $(isn_c, j) \in assoc$ then for all non-SYN segments $p \in$ *in-transit*$_{sc}$, $sn(p) < ack_c$.

2. If $mode_s \in$ {`close-wait`, `closing`, `last-ack`, `timed-wait`} and there exists i, such that $i = isn_c^s \ \wedge \ (i, isn_s) \in$ *estb-pairs* then for all non-SYN segments $p \in$ *in-transit*$_{cs}$, $sn(p) < ack_c$. ∎

Proof: In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') for Part 1 below. For Part 1 we know that a = *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c)* and a = *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c, FIN)* are not critical even though they may cause the pair (isn_c, isn_s) to be added to *assoc*. The pair is added if $s.mode_s$ = `syn-rcvd` $\wedge [ack_c] = s.sn_s + 1$. However, if these conditions are true in state s, then Invariant 7.49 tell us that $s.mode_c \notin$ {`close-wait`, `closing`, `last-ack`, `timed-wait`}, and since these steps do not change $mode_c$, we know the premise does not become true.

1. a = *receive-seg*$_{sc}$*(sn_s, ack_s, msg_s, FIN)*

   This step may cause the premise of Part 1 to go from false to true if $[sn_s] \geq ack_c$. If this change happens, then ack_c is also assigned $[sn_s] + 1$. From Invariant 7.62 we know that in this situation all segments $p \in$ *in-transit*$_{sc}$ have $sn(p) \leq [sn_s]$. Therefore, Part 1 holds after this step.

<u>a = *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and a = *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)*</u>

These steps may cause the premise of Part 1 to go from false to true, or the consequence of Part 1 to go from true to false. These steps are only enabled if $s.mode_s \in$ *sync-states*. If the steps cause the premise of the invariant to become true then by Invariants 7.38 and 7.44 we know $(s.isn_c, s.isn_s) \in s.assoc$. Therefore, by Invariant 7.57 we know that $sn_s < ack_c$, so Part 1 holds in this situation. If the steps cause the consequence of the Part 1 to go from true to false then again by Invariants 7.38, 7.44, and 7.57 we know the premise must be false.

2. We consider the critical actions for Part 2 below. We know that a = *receive-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)* is not critical even though this step may cause (isn_c, isn_s) to be in *estb-pairs* if $mode_c$ = `syn-rcvd`. However, we know from Invariant 7.19 that if this is the case in state s, then $mode_s \notin$ *sync-states*, so the premise does not become true after this step.

   <u>a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*</u>

   The proof for this case is symmetric to the case for a = *receive-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)* of Part 1.

   <u>a = *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and a = *send-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*</u>

   These steps may cause the premise of Part 2 to go from false to true, or the consequence of Part 2 to go from true to false. These steps are only enabled if $s.mode_c \in$ *sync-states*. From Invariant 7.34 we know that if $mode_c \in$ *sync-states* and the premise of the Part 2 is true, then $(isn_c, isn_s) \in$ *estb-pairs*. Therefore, by Invariant 7.57 we know that $sn_c < ack_s$, so Part 1 holds in this situation. If the steps cause the consequence of the Part 2 to go from true to false then again by Invariants 7.34 and 7.57 we know the premise must be false. ■

Invariant 7.66

1. If $mode_c \in \{$`close-wait`, `closing`, `timed-wait`$\}$ and there exists j such that $(isn_c, j) \in$

assoc then $push\text{-}data_c = \texttt{true} \vee rcv\text{-}buf_c = \epsilon$.

2. If If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{timed-wait}\}$ there exists i such that $(i, isn_s) \in estb\text{-}pairs \wedge isn_c = isn_c^s$ then $push\text{-}data_s = \texttt{true} \vee rcv\text{-}buf_s = \epsilon$.

Proof: In the initial state $mode_s$ and mode$_c$ are `closed`, so the invariant holds. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s)}$

   This step may cause the consequence of Part 1 to go from true to false, but only if $[sn_s] = ack_c$. We know from Invariant 7.65 that if the premise of Part 1 is true, then for all $p \in in\text{-}transit_{sc}$, $sn(p) < ack_c$. Therefore, if $[sn_s] = ack_c$, then the premise must be false, so Part 1 holds after this step.

   $\underline{a = receive\text{-}seg_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)}$

   This step may cause the premise of Part 1 to go from false to true. However, this step also sets $push\text{-}data_c$ to `true`. Therefore, Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.67

1. If $mode_c = \texttt{last-ack}$ and there exists j such that $(isn_c, j) \in assoc$ then $rcv\text{-}buf_c = \epsilon$.

2. If $mode_s = \texttt{last-ack}$ and there exists i, such that $i = isn_c^s \wedge (i, isn_s) \in estb\text{-}pairs$ then $rcv\text{-}buf_s = \epsilon$. ■

Proof: In the initial state $mode_s$ and mode$_c$ are `closed`, so the invariant holds. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = prepare\text{-}msg_c}$

   This step may change the premise of Part 1 from false to true. This change happens if $s.mode_c = \texttt{close-wait} \wedge \neg s.push\text{-}data_c$. However, if these conditions are true in state s, then from Invariant 7.66, we know that $rcv\text{-}buf_c = \epsilon$, so Part 1 holds after this step.

$\underline{a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s)}$

This step may cause the consequence of Part 1 to go from true to false, but only if $[sn_s] = ack_c$. However, Invariant 7.65 tells us that if this is true, then the premise of Part 1 must be false, so Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 7.68

If $mode_s$ = `syn-rcvd` and there exists a non-SYN segment $p \in \textit{in-transit}_{cs}$ such that $ack(p) = sn_s + 1$ and there exists a FIN segment $q \in \textit{in-transit}_{cs}$ such that $(sn(q) \geq \max(ack_s, sn(p)+1) \vee (p = q \wedge sn(q) \geq ack_s))$ then $\textit{rcvd-close}_c = \texttt{true} \vee isn_c \neq isn_c^s$.

Proof: In the start state $mode_c$ is `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We know that $a = \textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c)$ is not critical because Invariant 7.1 tells us that there cannot be a segment $q \in \textit{in-transit}_{cs}$ with $sn(q) > [sn_c]$, which must be the case if this step makes the premise of the invariant true.

$\underline{a = \textit{receive-msg}_{cs}(SYN,\ sn_c)}$

This step may cause the consequence of the invariant to go from true to false, by assiging isn_c^s to the value of isn_c. However, Invariant 7.11 tells us that after this step there are no segments $p \in \textit{in-transit}_{cs}$ such that $ack(p) = sn_s + 1$, so we the invariant holds after this step.

$\underline{a = \textit{send-seg}_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$

This step adds a FIN segment to $\textit{in-transit}_{cs}$, but only if $mode_c \in \{$`fin-wait-1`, `closing`, `last-ack`$\}$, so by Invariant 7.13 we know the invariant holds after this step. ■

Invariant 7.69

1. If $mode_c \in \textit{sync-states}$ and there exists j such that $(isn_c, j) \in \textit{assoc}$ and there exists a FIN segment $p \in \textit{in-transit}_{sc}$ such that $sn(p) \geq ack_c$ then $\textit{rcvd-close}_s = \texttt{true} \vee isn_s \neq j$.

2. If $mode_s \in \textit{sync-states}$ and there exists i such that $(i, isn_s) \in \textit{estb-pairs} \wedge i = isn_c^s$ and there exists a FIN segment $p \in \textit{in-transit}_{cs}$ such that $sn(p) \geq ack_s$ then $\textit{rcvd-close}_c =$

$\mathtt{true} \vee isn_c \neq i$.

Proof: In the start state $mode_c$ and $mode_s$ are `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We know that $a = \textit{receive-seg}_{sc}(SYN, sn_s, ack_s)$ is not critical even though it may assign $mode_c$ to `estb`, and ack_c to $[sn_s] + 1$. These assignment are made if $s.mode_c =$ `syn-sent`. From Invariant 7.16 we know that if $s.mode_c =$ `syn-sent`, then $(isn_c, isn_s) \notin assoc$. Thus, the premise of the invariant does not become true after this step. Steps with $a = \textit{receive-seg}_{cs}(sn_c, ack_c, msg_c)$ and $a = \textit{receive-seg}_{cs}(sn_c, ack_c, msg_c, FIN)$ may cause (isn_c, isn_s) to be added to $assoc$ if $[ack_c] > sn_s$. However, these steps do not make the premise of Part 1 go from false to true, because from Invariant 7.23 we know that $ack_c \geq [ack_c]$, and from Invariant 7.1 we know $sn_s \geq sn(p)$ for all $p \in \textit{in-transit}_{sc}$. Therefore, if these steps cause (isn_c, isn_s) to be added to $assoc$ then we know there are no segments $p \in \textit{in-transit}_{sc}$ such that $sn(p) \geq ack_c$.

1. <u>$a = \textit{send-seg}_{sc}(sn_s, ack_s, msg_s, FIN)$</u>

   This step adds a FIN segment to $\textit{in-transit}_{sc}$, but only if $s.mode_s \in \{$`fin-wait-1`, `last-ack`, `closing`$\}$. From Invariant 7.13 we know that if $s.mode_s$ is in this set, then $s.\textit{rcvd-close}_s =$ `true`. Since this step does not change the value of $\textit{rcvd-close}_s$, we know Part 1 is true after this step.

2. For Part 2 we know that $a = \textit{receive-seg}_{sc}(SYN, sn_s, ack_s)$ is not critical even though this step may cause (isn_c, isn_s) to be in $\textit{estb-pairs}$ if $mode_c =$ `syn-rcvd`. However, we know from Invariant 7.19 that if this is the case in state s, then $mode_s \notin \textit{sync-states}$, so the premise does not become true after this step.

   <u>$a = \textit{send-seg}_{cs}(sn_c, ack_c, msg_c, FIN)$</u>

   Symmetric to the case for $a = \textit{send-seg}_{sc}(sn_s, ack_s, msg_s, FIN)$ of Part 1.

   <u>$a = \textit{receive-seg}_{cs}(sn_c, ack_c, msg_c)$ and $a = \textit{receive-seg}_{cs}(sn_c, ack_c, msg_c, FIN)$</u>

   These steps may cause $mode_s$ to be in $\textit{sync-states}$ if $s.mode_s =$ `syn-rcvd`. However, we know from Invariant 7.68, that if this is the case, then the consequence of Part 2 is also true. ∎

Invariant 7.70

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists j such that $(isn_c, j) \in assoc$ then $rcvd\text{-}close_s = \texttt{true} \vee isn_s \neq j$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists i such that $(i, isn_s) \in estb\text{-}pairs \wedge i = isn_c^s$ then $rcvd\text{-}close_c = \texttt{true} \vee isn_c \neq i$. ∎

Proof: In the initial state $mode_s$ and mode$_c$ are `closed`, so the invariant holds. We consider critical actions of the form (s, a, s') below. For Part 1 we know that a = *receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$)* and a = *receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$, FIN)* are not critical even though they may cause the pair (isn_c, isn_s) to be added to *assoc*. The pair is added if $s.mode_s = \texttt{syn-rcvd} \wedge [ack_c] = s.sn_s + 1$. However, if these conditions are true in state s, then Invariant 7.49 tell us that $s.mode_c \notin \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$, and since these steps do not change $mode_c$, we know the premise does not become true.

1. <u>a = *receive-seg$_{sc}$(sn$_s$, ack$_s$, msg$_s$, FIN)*</u>

   This step may cause the premise of the invariant to go from false to true. This change happens if $s.mode_c \in sync\text{-}states$ and $[sn_s] \geq s.ack_c$. Therefore, from Invariant 7.69, we know the consequence is also true, so Part 1 holds after this step.

2. For Part 2 we know that a = *receive-seg$_{sc}$(SYN, sn$_s$, ack$_s$)* is not critical even though this step may cause (isn_c, isn_s) to be in *estb-pairs* if $mode_c = \texttt{syn-rcvd}$. However, we know from Invariant 7.19 that if this is the case in state s, then $mode_s \notin sync\text{-}states$, so the premise does not become true after this step.

   <u>a = *receive-seg$_{cs}$(sn$_c$, ack$_c$, msg$_c$, FIN)*</u>

   The proof that Part 2 holds after this step is symmetric to the proof that Part 1 holds after a = *receive-seg$_{sc}$(sn$_s$, ack$_s$, msg$_s$, FIN)*. ∎

Appendix C

Invariance proofs for $\mathcal{BTCP}^h$

In this appendix we prove the invariants of $\mathcal{BTCP}^h$ presented in Chapter 8. In the statement of the invariants and in the proofs $i > j$ if and only if $i \in \{j+1, \ldots, j+(2^{31}-1)\}$, where the additions are modulo 2^{32}. As we did in Appendix B, we use the standard inductive technique for proving the invariants. That is, we show that the invariants hold for the start states and then show that for every step (s, a, s') of $\mathcal{TCP}^h$, if the invariant holds in state s then it also holds in state s'. In the proofs of the invariants from Chapter 8, we use other invariants of $\mathcal{BTCP}^h$ that we state and also prove in this appendix.

Invariant C.1

1. If $(p, t) \in \textit{in-transit}_{cs}$ then $now \leq t \leq now + \mu$.

2. If $(p, t) \in \textit{in-transit}_{sc}$ then $now \leq t \leq now + \mu$.

Proof: In the start state $\textit{in-transit}_{cs}$ and $\textit{in-transit}_{sc}$ are both empty, so the invariant holds for this state. We consider critical steps of the form (s, a, s') below.

1. $\underline{a = \textit{send-seg}_{cs}(p)}$

   These steps add segments to $\textit{in-transit}_{cs}$. However, the timestamp on the segment is $now + \mu$.

   $\underline{\nu(t')}$

   This is only enabled if for all segments $(p, t) \in \textit{in-transit}_{cs}$ $s.now + t' \leq t$. Therefore, if this step cause the consequence to go from true to false, then the premise must also be false.

2. The proof for Part 2 is symmetric to the proof for Part 1. ∎

Invariant C.2

1. For all $x \in \texttt{BN}$ $\mathit{lst\text{-}time\text{-}cc}_c(x) \leq \mathit{now}$.

2. For all $x \in \texttt{BN}$ $\mathit{lst\text{-}time\text{-}cc}_s(x) \leq \mathit{now}$.

3. For all $x \in \texttt{BN}$ $\mathit{lst\text{-}time\text{-}sn}_c(x) \leq \mathit{now}$.

4. For all $x \in \texttt{BN}$ $\mathit{lst\text{-}time\text{-}sn}_s(x) \leq \mathit{now}$.

Proof: Straightforward. ∎

Invariant C.3

1. $\mathit{first(tick_c)} = \mathit{last(tick_c)}$

2. $\mathit{first(tick_s)} = \mathit{last(tick_s)}$

3. $\mathit{first(tick_c)} \leq \mathit{now} + \mathit{clock\text{-}rate}$.

4. $\mathit{first(tick_s)} \leq \mathit{now} + \mathit{clock\text{-}rate}$.

5. If $\mathit{mode}_c \neq \texttt{rec}$ then $\mathit{now} \leq \mathit{last(tick_c)}$.

6. If $\mathit{mode}_s \neq \texttt{rec}$ then $\mathit{now} \leq \mathit{last(tick_s)}$.

Proof: Straightforward. ∎

Invariant C.4

1. If $\mathit{mode}_c \neq \texttt{closed}$ then $\mathit{con\text{-}strt\text{-}time}_c \leq \mathit{now}$.

2. If $\mathit{mode}_s \notin \{\texttt{closed}, \texttt{listen}\}$ then $\mathit{con\text{-}strt\text{-}time}_s \leq \mathit{now}$.

Proof: Straightforward. ∎

Invariant C.5

1. $\mathit{sn}_c \in \texttt{BN}$ if and only if $\mathit{mode}_c \neq \texttt{closed}$.

2. $\mathit{sn}_s \in \texttt{BN}$ if and only if $\mathit{mode}_s \notin \{\texttt{closed}, \texttt{listen}\}$.

Proof: Straightforward. ∎

Invariant C.6

1. For all segments $(p, t) \in \textit{in-transit}_{cs}$ $\textit{sn}(p) \neq \texttt{nil}$.

2. For all segments $(p, t) \in \textit{in-transit}_{sc}$ $\textit{sn}(p) \neq \texttt{nil}$.

Proof: Straightforward. ■

Invariant C.7

1. If $\textit{mode}_c \neq \texttt{closed} \wedge \textit{wait-t\_o}_c \neq \infty$ then $\textit{wait-t\_o}_c \leq \textit{now} + \textit{wt}$.

2. If $\textit{mode}_s \neq \texttt{closed} \wedge \textit{wait-t\_o}_s \neq \infty$ then $\textit{wait-t\_o}_s \leq \textit{now} + \textit{wt}$.

Proof: Straightforward. ■

Invariant C.8

If $\textit{mode}_c = \texttt{rec}$ then $\textit{first(recov}_c\textit{)} > t + \mu + \textit{rt} + \textit{wt}$ for any segment $(p, t) \in \textit{in-transit}_{cs}$.

Proof: In the start state $\textit{mode}_c = \texttt{closed}$ so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$\underline{a = \textit{crash}_c}$

This step may cause the premise of Part 1 to go from false to true. However, in this step $\textit{first(recov}_c\textit{)}$ is set to $\textit{now} + \textit{qt}$, and from Invariant C.1 we know that for any segment $(p, t) \in \textit{in-transit}_{cs}$, $t \leq \textit{now} + \mu$. Therefore, since $\textit{qt} > \textit{wt} + \textit{rt} + 2\mu$, Part 1 holds after this step.

$\underline{a = \textit{send-seg}_{cs}(p)}$

These steps may cause the consequence of Part 1 to go from true to false, but they are only enabled if $\textit{mode}_c \neq \texttt{rec}$. ■

Invariant C.9

If $\textit{mode}_s = \texttt{syn-rcvd} \wedge \textit{wait-t\_o}_s = \infty$ then $\textit{last(response}_s\textit{)} \leq \textit{now} + \textit{rt}$.

Proof: In the start state $\textit{mode}_s = \texttt{closed}$ so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$\underline{a = \textit{receive-seg}_{cs}(\textit{SYN}, \textit{sn}_c)}$

This step may cause the premise of the invariant to go from false to true. However in this step $\textit{last(response}_c\textit{)}$ is set to $\textit{now} + \textit{rt}$, so the invariant holds after this step.

$a = send\text{-}seg_{cs}(p)$

These steps may cause the consequence of the invariant to go from true to false. In the case where $p = (SYN, sn_c)$, after that step $wait\text{-}t\_o_c \neq \infty$. After the other steps $mode_c \neq$ `syn-rcvd`, so the invariant holds. ■

Invariant C.10

If $mode_c =$ `rec` $\wedge$ $mode_s =$ `syn-rcvd` $\wedge$ $wait\text{-}t\_o_s = \infty$ then $first(recov_c) > last(response_s) + wt + \mu$.

Proof: In the start state $mode_c =$ `closed` so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$a = crash_c$

This step may cause the premise of the invariant to go from false to true. From Invariant C.9 we know that if $mode_s =$ `syn-rcvd` $\wedge$ $wait\text{-}t\_o_s = \infty$ then $last(response_s) \leq now + rt$. Therefore, since $first(recov_c)$ is set to $now + qt$ in this step, the invariant holds.

$a = receive\text{-}seg_{cs}(SYN,\ sn_c)$

This step may cause the premise of the invariant to go from false to true. Let (p, t) be the segment received in this step. From Invariant C.8 we know that if $mode_c =$ `rec` then $first(recov_c) > t + \mu + rt + wt$. Since $last(response_s)$ is set to $now + rt$ in this step, and from Invariant C.1 we know $t \geq now$, the invariant holds after this step. ■

Invariant C.11

If $mode_c =$ `rec` $\wedge$ $mode_s =$ `syn-rcvd` $\wedge$ $wait\text{-}t\_o_s = \infty$ then $first(recov_c) > now + wt + \mu$.

Proof: In the start state $mode_c =$ `closed` so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$a = crash_c$

This step may cause the premise of the invariant to go from false to true. Since $first(recov_c)$ is set to $now + qt$ in this step, the invariant clearly holds.

$a = receive\text{-}seg_{cs}(SYN,\ sn_c)$

This step may cause the premise of the invariant to go from false to true. Let (p, t) be the segment received in this step. From Invariant C.8 we know that if $mode_c =$ `rec` then

$\mathit{first}(\mathit{recov}_c) > t + \mu + rt + wt$. Since by Invariant C.1, $t \geq \mathit{now}$, we know the invariant holds after this step.

$a = \nu(t')$

This step may cause the consequence of the invariant to go from true to false. It is only enabled if $s.\mathit{now} + t' \leq \mathit{last}(\mathit{response}_s)$. From Invariant C.10 we know that if $\mathit{mode}_c =$ `rec` $\wedge$ $\mathit{mode}_s =$ `syn-rcvd` $\wedge$ $\mathit{wait\text{-}t\_o}_s = \infty$ then $\mathit{first}(\mathit{recov}_c) > \mathit{last}(\mathit{response}_s) + wt + \mu$. Therefore, if this step causes the consequence of the invariant to be false, then the premise must also be false. ■

Invariant C.12

If $\mathit{mode}_c =$ `rec` $\wedge$ $\mathit{mode}_s =$ `syn-rcvd` $\wedge$ $\mathit{wait\text{-}t\_o}_s \neq \infty$ then $\mathit{first}(\mathit{recov}_c) > \mathit{wait\text{-}t\_o}_s + \mu$.

Proof: In the start state $\mathit{mode}_c =$ `closed` so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$a = \mathit{crash}_c$

This step may cause the premise of the invariant to go from false to true. However, from Invariant C.7 we know if $\mathit{wait\text{-}t\_o}_s \neq \infty$ then $\mathit{wait\text{-}t\_o}_s \leq \mathit{now} + wt$. Since $\mathit{first}(\mathit{recov}_c)$ is set to $\mathit{now} + qt$ in this step, the invariant holds after this step.

$a = \mathit{send\text{-}seg}_{sc}(SYN,\ sn_s,\ ack_s)$

This step may cause the premise of the invariant to go from false to true by setting $\mathit{wait\text{-}t\_o}_s$ to $\mathit{now} + wt$, if $s.\mathit{mode}_s =$ `syn-rcvd` $\wedge$ $s.\mathit{wait\text{-}t\_o}_s = \infty$. However, from Invariant C.11 we know that if these conditions are true in state s, then $s.\mathit{first}(\mathit{recov}_c) > s.\mathit{now} + wt + \mu$. Therefore, since this step does not change $\mathit{first}(\mathit{recov}_c)$ or now, the invariant holds after this step.

$a = \mathit{receive\text{-}seg}_{cs}(p)$

These steps may cause the consequence of the invariant to go from true to fales by assigning $\mathit{wait\text{-}t\_o}_s$ to ∞. However, after each of these steps the premise is also clearly false, so the invariant holds. ■

Invariant C.13

If $\mathit{mode}_c =$ `rec` $\wedge$ $\mathit{mode}_s =$ `syn-rcvd` $\wedge$ $\mathit{now} \leq \mathit{wait\text{-}t\_o}_s$ then $\mathit{first}(\mathit{recov}_c) > \mathit{now} + \mu$

Proof: In the start state $mode_c = \texttt{closed}$ so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$\underline{a = crash_c}$

This step may cause the premise of the invariant to go from false to true. Since *first(recov$_c$)* is set to $now + qt$ in this step, the invariant clearly holds.

$\underline{a = receive\text{-}seg_{cs}(SYN,\ sn_c)}$

This step may cause the premise of the invariant to go from false to true. Let (p, t) be the segment received in this step. From Invariant C.8 we know that if $mode_c = \texttt{rec}$ then *first(recov$_c$)* $> t + \mu + rt + wt$. Since by Invariant C.1, $t \geq now$, we know the invariant holds after this step.

$\underline{a = \nu(t')}$

This step may cause the consequence of the invariant to go from true to false. However, from Invariant C.12 we know that if $mode_c = \texttt{rec} \ \wedge \ mode_s = \texttt{syn-rcvd} \ \wedge wait\text{-}t\_o_s \neq \infty$ then *first(recov$_c$)* $> wait\text{-}t\_o_s + \mu$. Therfore, if this step causes $now + \mu$ to be greater than or equal to *first(recov$_c$)* then the premise of the invariant must also be false. ∎

Invariant C.14

If $mode_c = \texttt{rec}$ then *first(recov$_c$)* $> t$ for any SYN segment $(p, t) \in in\text{-}transit_{sc}$.

Proof: In the start state $mode_c = \texttt{closed}$ so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$\underline{a = crash_c}$

This step may cause the premise of the invariant to go from false to true. However, in this step *first(recov$_c$)* is set to $now + qt$, and from Invariant C.1 we know that for any segment $(p, t) \in in\text{-}transit_{sc}$, $t \leq now + \mu$. Therefore, since $qt > wt + rt + 2\mu$, the invariant holds after this step.

$\underline{a = send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)}$

This step may cause the consequence of the invariant to go from true to false by adding a SYN segment (p, t) where $t = now + \mu$ and $t \geq$ *first(recov$_c$)*. However, it is only enabled if $s.mode_s = \texttt{syn-rcvd} \ \wedge s.now \leq s.wait\text{-}t\_o_s$. From Invariant C.13 we know that if $mode_s = \texttt{syn-rcvd} \ \wedge now \leq wait\text{-}t\_o_s \ \wedge mode_c = \texttt{rec}$ then *first(recov$_c$)* $> now + \mu$.

Therefore, if this step causes the consequence of the invariant to be false, then the premise must also be false. ∎

Invariant C.15

1. If $mode_c = \texttt{rec} \wedge now \geq \mathit{first(recov_c)}$ then $\mathit{in\text{-}transit}_{cs} = \emptyset$.

2. If $mode_c = \texttt{rec} \wedge now \geq \mathit{first(recov_c)}$ then there are no SYN segments in $\mathit{in\text{-}transit}_{sc}$.

Proof: In the start state $mode_c = \texttt{closed}$ so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = \nu(t)}$

   This step may cause the premise of Part 1 to go from false to true. However, from Invariant C.8 we know that $\mathit{first(recov_c)} > t$ for any segment $(p, t) \in \mathit{in\text{-}transit}_{cs}$, and from Invariant C.1 we also know that $now \leq t$ for any segment $(p, t) \in \mathit{in\text{-}transit}_{cs}$. Therefore, there cannot be any segment in $\mathit{in\text{-}transit}_{cs}$ if $now \geq \mathit{first(recov_c)}$, so Part 1 holds after this step.

   $\underline{a = \mathit{send\text{-}seg}_{cs}(p)}$

   These steps may cause the consequence of Part 1 to go from true to false, but they are only enabled if $mode_c \neq \texttt{rec}$.

2. $\underline{a = \nu(t)}$

   This step may cause the premise of Part 2 to go from false to true. However, from Invariant C.14 we know that $\mathit{first(recov_c)} > t$ for any SYN segment $(p, t) \in \mathit{in\text{-}transit}_{sc}$, and from Invariant C.1 we also know that $now \leq t$ for any segment $(p, t) \in \mathit{in\text{-}transit}_{sc}$. Therefore, there cannot be any SYN segments in $\mathit{in\text{-}transit}_{sc}$ if $now \geq \mathit{first(recov_c)}$, so Part 2 holds after this step.

   $\underline{a = \mathit{send\text{-}seg}_{sc}(SYN,\ sn_s,\ ack_s)}$

   This step may cause the consequence of Part 2 to go from true to false. This step is enabled if $s.mode_s = \texttt{syn-rcvd} \wedge now \leq s.\mathit{wait\text{-}t\_o}_s$. From Invariant C.13 we know that $s.mode_s = \texttt{syn-rcvd} \wedge now \leq s.\mathit{wait\text{-}t\_o}_s \wedge s.mode_c = \texttt{rec}$ then $\mathit{first(recov_c)} > now + \mu$. Thus, if this step is enabled, the premise of Part 2 must also be false. ∎

Invariant C.16

1. If $mode_c = \texttt{closed} \wedge now > \mathit{first(open_c)}$ then $\mathit{in\text{-}transit}_{cs} = \emptyset$.

2. If $mode_s = \texttt{closed} \wedge now > \mathit{first(open_s)} \vee mode_s = \texttt{listen}$ then $\mathit{in\text{-}transit}_{sc} = \emptyset$.

Proof: Straightforward given Invariants C.22 and C.1. ■

Invariant C.17

1. If $mode_c = \texttt{rec}$ then $\mathit{lst\text{-}crash\text{-}time}_c = \mathit{first(recov_c)} - qt$.

2. If $mode_s = \texttt{rec}$ then $\mathit{lst\text{-}crash\text{-}time}_s = \mathit{first(recov_s)} - qt$.

3. If $mode_c \neq \texttt{rec}$ then $\mathit{lst\text{-}crash\text{-}time}_c \leq now - qt$.

4. If $mode_s \neq \texttt{rec}$ then $\mathit{lst\text{-}crash\text{-}time}_s \leq now - qt$.

Proof: Straightforward. ■

Invariant C.18

1. If $mode_c = \texttt{syn-sent}$ then $\mathit{lst\text{-}time\text{-}cc}_c(sn_c) > \mathit{lst\text{-}crash\text{-}time}_c$.

2. If $mode_s = \texttt{syn-rcvd}$ then $\mathit{lst\text{-}time\text{-}cc}_s(sn_s) > \mathit{lst\text{-}crash\text{-}time}_s$.

In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s').

1. <u>$a = \mathit{send\text{-}msg(open,\ m,\ close)}$</u>

   This step may cause the premise of Part 1 to go from true to false. However, from Invariant C.35 we know that $s'.\mathit{lst\text{-}time\text{-}cc}_c(s'.sn_c) \geq s'.now - rt - \mathit{clock\text{-}rate}$, and from Invariant C.17 we know that $s'.\mathit{lst\text{-}crash\text{-}time}_c \leq s'.now - qt$. Since $qt > rt + \mathit{clock\text{-}rate}$, we now Part 1 holds after this step.

   <u>$a = \mathit{crash}_c$</u>

   This step may cause the consequence of Part 1 to go from true to false by setting $\mathit{lst\text{-}crash\text{-}time}_c$ to now. However, this step also sets $mode_c$ to `rec`, so Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant C.19

If $mode_c \neq \texttt{rec} \wedge \mathit{lst\text{-}crash\text{-}time}_c \geq \mathit{lst\text{-}time\text{-}cc}_c(x)$ then there are no SYN segments $(p, t) \in \mathit{in\text{-}transit}_{cs}$ with $sn(p) = x$.

In the start state $\mathit{in\text{-}transit}_{cs}$ is empty, so the consequence of the invariant is true, which means the invariant holds for this state. We consider critical steps of the form (s, a, s') below.

$a = \mathit{recover}_c$

This step may cause the premise of the invariant to go from false to true. This step is enabled if $s.mode_c = \texttt{rec} \wedge s.now \geq \mathit{first}(\mathit{recov}_c)$. From Invariant C.15 we know that $s.\mathit{in\text{-}transit}_{cs}$ is empty.

$a = \mathit{send\text{-}seg}_{cs}(SYN,\ sn_c)$

This step adds a SYN segment to $\mathit{in\text{-}transit}_{cs}$ if $mode_c = \texttt{syn-sent} \wedge now \leq \mathit{wait\text{-}t\_o}_c$. It may cause the consequence of the invariant to go from true to false if $sn_c = x$. However, from Invariant C.18 we know that if $mode_c = \texttt{syn-sent}$ then $\mathit{lst\text{-}time\text{-}cc}_c(sn_c) > \mathit{lst\text{-}crash\text{-}time}_c$. Therefore, if the consequence of the invariant becomes false after this step, then the premise must also be false. ■

Invariant C.20

If $mode_c \neq \texttt{rec} \wedge \mathit{lst\text{-}crash\text{-}time}_c \geq \mathit{lst\text{-}time\text{-}cc}_c(ack_s - 1)$ then $\neg(mode_s = \texttt{syn-rcvd} \wedge now \leq \mathit{wait\text{-}t\_o}_s)$.

Proof: In the start state $mode_s = \texttt{closed}$, so the consequence of the invariant is true, which means the invariant holds for this state. We consider critical steps of the form (s, a, s') below.

$a = \mathit{recover}_c$

This step may cause the premise of the invariant to go from false to true. This step is enable if $s.mode_c = \texttt{rec} \wedge s.now \geq \mathit{first}(\mathit{recov}_c)$. From Invariant C.13 we know that if $mode_c = \texttt{rec} \wedge mode_s = \texttt{syn-rcvd} \wedge now \leq \mathit{wait\text{-}t\_o}_s$ then $\mathit{first}(\mathit{recov}_c) > now + \mu$. Therefore, if this step is enabled, the consequence of the invariant must also be true.

$a = \mathit{receive\text{-}seg}_{cs}(SYN,\ sn_c)$

This step may cause the consequence of the invariant to go from true to false. Let (p, t) be the segment received in this step. After the step $s'.ack_s - 1 = sn(p)$. However, we know by Invariant C.19 that if $mode_c \neq \texttt{rec} \wedge \textit{lst-crash-time}_c \geq \textit{lst-time-cc}_c(x)$ then there are no SYN segments in $\textit{in-transit}_{cs}$ with $sn(p) = x$. Therefore, if this step causes the consequence to be false, the premise must also be false. ∎

Invariant C.21

If $mode_c \neq \texttt{rec} \ \wedge \textit{lst-crash-time}_c \geq \textit{lst-time-cc}_c(x-1)$ then there are no SYN segments (p,t) in $\textit{in-transit}_{sc}$ with $ack(p) = x$.

Proof: In the start state $mode_s = \texttt{closed}$, so the consequence of the invariant is true, which means the invariant holds for this state. We consider critical steps of the form (s, a, s') below.

$a = \textit{recover}_c$

This step may cause the premise of the invariant to go from false to true. This step is enable if $s.mode_c = \texttt{rec} \ \wedge s.now \geq \textit{first(recov}_c\textit{)}$. From Invariant C.15 we know that there are no SYN segments in $s.\textit{in-transit}_{sc}$. Therefore, if this step is enabled, the consequence of the invariant must also be true.

$a = \textit{send-seg}_{sc}\textit{(SYN, } sn_s\textit{, } ack_s\textit{)}$

This step may cause the consequence of the invariant to go from true to false. It is enabled if $mode_s = \texttt{syn-rcvd} \ \wedge now \leq \textit{wait-t\_o}_s$. From Invariant C.20 we know that if this step is enabled, then the premise of the invariant must be false. ∎

Invariant C.22

1. If $mode_c \in \{\texttt{rec}, \texttt{closed}\}$ then for all segments $p \in \textit{in-transit}_{cs}$, $t \leq \textit{first(open}_c\textit{)}$.

2. If $mode_s \in \{\texttt{rec}, \texttt{closed}\}$ then for all segments $p \in \textit{in-transit}_{sc}$, $t \leq \textit{first(open}_s\textit{)}$.

Proof: . In the start state both $\textit{in-transit}_{cs}$ and $\textit{in-transit}_{sc}$ are empty, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

1. $a = \textit{receive-seg}_{sc}\textit{(}sn_s\textit{, } ack_s\textit{, } msg_s\textit{)}$, $a = \textit{shut-down}_c$, and $\textit{time-out}_c$

   These steps may cause the premise of Part 1 to go from false to true. However, in all of these steps $\textit{first(open}_c\textit{)}$ is set to $now + \mu$. Since by Invariant C.1 we know $t \leq now + \mu$, we know the consequence is also true. Thus, Part 1 holds after these steps.

$\underline{a = crash_c}$

This step may cause the premise of Part 1 to go from false to true. However, in this step *first(open$_c$)* is set to $now + qt$. Since by Invariant C.1 we know $t \leq now + \mu$, we know the consequence is also true.

$\underline{a = send\text{-}seg_{cs}(p)}$

Any step that adds a segment to $in\text{-}transit_{cs}$, may cause the consequence of Part 1 to go from true to false. However, since these actions are only enabled if $mode_c \neq$ `closed`, the premise must also be false in these states.

2. The proof of Part 2 is symmetric. ■

Invariant 8.1

1. If $mode_c =$ `syn-sent` and for $(p, t) \in in\text{-}transit_{cs}$, $t - \mu \geq con\text{-}strt\text{-}time_c$ then $sn_c = sn(p)$.

2. If $mode_s =$ `syn-rcvd` and for $(p, t) \in in\text{-}transit_{sc}$, $t - \mu \geq con\text{-}strt\text{-}time_s$ then $sn_s = sn(p)$.

Proof: In the start state $mode_s =$ `closed` and $mode_c =$ `closed`, so the invariant holds in this state. We consider the critical steps of the form (s, a, s') below.

1. $\underline{a = send\text{-}msg_c(open,\ m,\ close)}$

   This step may cause the consequence of Part 1 to go from true to false by assigning sn_c to $s.clock\text{-}counter_c$. This assignment is made only if $s.mode_c =$ `closed`. Also if the assignment is made in this step, then $con\text{-}strt\text{-}time_c$ is also assigned *now*. Furthermore, we know that this step is only enabled if $now >$ *first(open$_c$)*. Therefore, by Invariant C.22 we know that for all $(p, t) \in s'.in\text{-}transit_{cs}$, $t < s'.con\text{-}strt\text{-}time_c$. Thus, the premise is false after this step also.

   $\underline{a = send\text{-}seg_{cs}(p)}$

   These steps can make the premise of Part 1 go from false to true by add a segment (p, t) to $in\text{-}transit_{cs}$ with $t - \mu \geq con\text{-}strt\text{-}time_c$. However, for any such segment, $sn(p) = sn_c$.

$\underline{a = \textit{prepare-msg}_c}$

This step may cause the consequence of Part 1 to go from true to false by incrementing sn_c. However, this step is only enabled if $mode_c \neq$ `syn-sent`, so Part 1 holds after this step.

2. The proof for Part 2 is symmetric. ∎

Invariant 8.2

1. If $mode_c =$ `syn-sent` $\wedge$ $\textit{new-isn}_c$ then for all $(p, t) \in \textit{in-transit}_{cs}$, $t - \mu < \textit{con-strt-time}_c$.

2. If $mode_s =$ `syn-rcvd` $\wedge$ $\textit{new-isn}_s$ then for all $(p, t) \in \textit{in-transit}_{sc}$, $t - \mu < \textit{con-strt-time}_s$.

Proof: In the start state $mode_s =$ `closed` and $mode_c =$ `closed`, so the invariant holds in this state. We consider the critical steps of the form (s, a, s') below.

1. $\underline{a = \textit{send-msg}_c\textit{(open, m, close)}}$

   This step may cause the premise of Part 1 to go from false to true. This assignment is made only if $s.mode_c =$ `closed`. Also if the assignment is made in this step, then $\textit{con-strt-time}_c$ is also assigned *now*. Furthermore, we know that this step is only enabled if $now > \textit{first(open}_c\textit{)}$. Therefore, by Invariant C.22 we know that for all $(p, t) \in s'.\textit{in-transit}_{cs}$, $t < s'.\textit{con-strt-time}_c$. Therefore, Part 1 holds after this step.

   $\underline{a = \textit{send-seg}_{cs}\textit{(p)}}$

   These steps can make the consequence of Part 1 go from true to false, but these steps also set $\textit{new-isn}_c$ to `false`, so the premise is also false.

2. The proof of Part 2 is symmetric. ∎

Invariant 8.3

1. If $mode_c \neq$ `closed` then for all segments $(p, t) \in \textit{in-transit}_{cs}$, $t \geq \textit{con-strt-time}_c + \mu$.

2. If $mode_s \neq$ `closed` then for all segments $(p, t) \in \textit{in-transit}_{sc}$, $t \geq \textit{con-strt-time}_s + \mu$.

Proof: In the start state $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = \textit{send-msg}_c(\textit{open, m, close})}$

   This step can make the premise of Part 1 to go from false to true. This change occurs if $now > \textit{first(open}_c)$, and if this change occurs $\textit{con-strt-time}_c$ is set to now. From Invariant C.22 we know that for all segment $(p, t) \in \textit{in-transit}_{cs}$, $t \leq \textit{first(open}_c)$, and from Invariant C.1 we know that $now \leq t$. These facts couple with the fact that the premise only goes from true to false in this step if $now > \textit{first(open}_c)$, means there cannot be a segment $(p, t) \in s.\textit{in-transit}_{cs}$. Therefore, Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 8.4

1. If $mode_s \in \textit{sync-states}$ and there exists segment $(p, t) \in \textit{in-transit}_{cs}$ such that $sn(p) = sn_c$ and $sn(p) = ack_s$ then $t - \mu \geq \textit{con-strt-time}_c$.

2. If $mode_c \in \textit{sync-states}$ and there exists segment $(p, t) \in \textit{in-transit}_{sc}$ such that $sn(p) = sn_s$ and $sn(p) = ack_c$ then $t - \mu \geq \textit{con-strt-time}_s$.

Proof: In the start state both $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. From Invariant C.6 we know that for any segment on a channel $sn(p) \neq$ `nil`, and from Invariant C.5 we know that if $sn_c \neq$ `nil` then $mode_c \neq$ `closed` and if $mode_s \notin$ $\{$`closed`, `listen`$\}$ then $sn_c \neq$ `nil`. Therefore, if $sn(p) = sn_c$ then $mode_c \neq$ `closed` and if $sn(p) = sn_s$ then $mode_s \neq$ `closed`. From Invariant 8.3 we know that if $mode_c \neq$ `closed` then for all segments $(p, t) \in \textit{in-transit}_{cs}$, $t \geq \textit{con-strt-time}_c + \mu$, and if $mode_s \neq$ `closed` then for all segments $(p, t) \in \textit{in-transit}_{sc}$, $t \geq \textit{con-strt-time}_s + \mu$. Therefore, the invariant holds. ■

Invariant C.23

1. If $mode_c =$ `closed` then for all $x \in$ `BN`, $\textit{lst-time-sn}_c(x) \leq \textit{first(open}_c) - \mu$.

2. If $mode_s =$ `closed` then for all $x \in$ `BN`, $\textit{lst-time-sn}_s(x) \leq \textit{first(open}_s) - \mu$.

Proof: Straightforward. ■

Invariant C.24

1. If $mode_c =$ `syn-sent` then any segment in $\textit{in-transit}_{cs}$ is a SYN segment.

2. If $mode_s$ = `syn-rcvd` then any segment in $in\text{-}transit_{sc}$ is a SYN segment.

Proof: If the start state $mode_c$ = `closed` and $mode_s$ = `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

1. $\underline{a = \mathit{send\text{-}msg_c(open,\ m,\ close)}}$

   This step may cause the premise of Part 1 to go from false to true if $s.mode_c$ = `closed` $\wedge$ $now > \mathit{first(open_c)}$. From Invariant C.22 we know that if $mode_c$ = `closed` then for all segments $(p, t) \in in\text{-}transit_{cs}$, $fopenc \geq t$. From Invariant C.1 we know that $now \leq t$ for all segments $(p, t) \in in\text{-}transit_{cs}$. Therefore, if this step causes the premise of Part 1 to be true, then there are no segments in $s'.in\text{-}transit_{cs}$.

   $\underline{a = \mathit{send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)} \text{ and } a = \mathit{send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}}$

   These steps may cause the consequence of Part 1 to go from true to false by adding a non-SYN segment to $in\text{-}transit_{cs}$. However, these steps are only enabled if $mode_c \neq$ `syn-sent`.

2. The proof for Part 2 is symmetric to the proof for Part 1. ∎

Invariant C.25

1. If $mode_c$ = `rec` then for all $x \in$ `BN`, $lst\text{-}time\text{-}cc_c(x) \leq \mathit{first(recov_c)} - qt$.

2. If $mode_s$ = `rec` then for all $x \in$ `BN`, $lst\text{-}time\text{-}cc_s(x) \leq \mathit{first(recov_s)} - qt$.

Proof: In the start state both $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = crash_c}$

   This step makes the premise of Part 1 go from false to true. In this step $\mathit{first(recov_c)}$ is assgined to $now + qt$. From Invariant C.2 we know that for all $x \in$ `BN` $lst\text{-}time\text{-}cc_c(x) \leq now$. Therefore, Part 1 holds after this step.

   $\underline{a = recover_c \text{ and } a = clock\text{-}counter\text{-}tick_c}$

   These steps may cause the consequence of Part 1 to go from true to false. However, after these steps $mode_c \neq$ `rec`, so Part 1 holds.

2. The proof for Part 2 is symmetric. ■

Invariant C.26

1. For all $x \in \mathtt{BN}$, *first(tick$_c$)* $-$ *clock-rate* $\geq$ *lst-time-cc$_c$*(x).

2. For all $x \in \mathtt{BN}$, *first(tick$_s$)* $-$ *clock-rate* $\geq$ *lst-time-cc$_s$*(x).

Proof: In the start state for all $x \in \mathtt{BN}$ *lst-time-cc$_c$*(x) and *lst-time-cc$_s$*(x) have the value 0, and *first(tick$_c$)* and *first(tick$_s$)* have the value *clock-rate*, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = recover_c \text{ and } a = clock\text{-}counter\text{-}tick_c}$

   These steps assign *lst-time-cc$_c$*(*clock-counter$_c$*) to *now*. However, these steps also assign *first(tick$_c$)* to *now* + *clock-rate*. Therefore, Part 1 holds after these steps.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant C.27

1. If *mode$_c$* $\neq$ `closed` then for all $x \in \mathtt{BN}$, *first(prep-msg$_c$)* $-$ *data-rate* $\geq$ *lst-time-sn$_c$*(x).

2. f *mode$_c$* $\notin$ {`closed`, `listen`} then for all $x \in \mathtt{BN}$, *first(prep-msg$_s$)* $-$ *data-rate* $\geq$ *lst-time-sn$_s$*(x).

In the start state *mode$_c$* and *mode$_s$* are equal to `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below. **Proof:** Straightforward. ■

Invariant C.28

1. If *mode$_c$* = `rec` then for all $x \in \mathtt{BN}$, *lst-crash-time$_c$* $\geq$ *lst-time-cc$_c$*(x).

2. If *mode$_s$* = `rec` then for all $x \in \mathtt{BN}$, *lst-crash-time$_s$* $\geq$ *lst-time-cc$_s$*(x).

Proof: Straightforward. ■

Invariant C.29

1. If *clock-counter$_c$* $= x \;\wedge\;$ *lst-time-cc$_c$*$(x) \neq 0$ and for $i \in \mathtt{BN}$, *lst-crash-time$_c$* $<$ *lst-time-cc$_c$*$(x-i)$ then *lst-time-cc$_c$*(x) $-$ *lst-time-cc$_c$*$(x-i) \geq i \times$ *clock-rate*.

2. If *clock-counter$_s$* $= x \;\wedge\;$ *lst-time-cc$_s$*$(x) \neq 0$ and for $i \in \mathtt{BN}$, *lst-crash-time$_s$* $<$ *lst-time-cc$_s$*$(x-i)$ then *lst-time-cc$_s$*(x) $-$ *lst-time-cc$_s$*$(x-i) \geq i \times$ *clock-rate*.

Proof: For this invariant we have two levels of induction. The first level is induction on i, and the second level is the induction on the steps of $\mathcal{BTCP}^h$. The base case of the induction on i is for $i = 0$. The invariant clearly holds for this case. For the inductive case we assume that the Invariant holds for j and show it holds for $j + 1$. We show it holds for $j + 1$ by induction on the steps of $\mathcal{BTCP}^h$.

In the start state for all x $\mathit{lst\text{-}time\text{-}cc}_c(x)$ and $\mathit{lst\text{-}time\text{-}cc}_s(x)$ have the value 0, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below. The step with $a = \mathit{recover}_c$ is not critical for Part 1 because if this step is enabled only if $\mathit{mode}_c =$ `rec`, which by Invariant C.28 means for all $x \in \mathtt{BN}$, $\mathit{lst\text{-}crash\text{-}time}_c \geq \mathit{lst\text{-}time\text{-}cc}_c(x)$. The step with $a = \mathit{recover}_s$ is not critical for Part 2 for symmetric reasons.

1. $\underline{a = \mathit{clock\text{-}counter\text{-}tick}_c}$

   This step may cause the premise of the Part 1 to go from false to true by assigning $\mathit{clock\text{-}counter}_c$ to the value x and $\mathit{lst\text{-}time\text{-}cc}_c(x)$ to now. This step is enabled if $\mathit{now} \geq \mathit{first(tick_c)}$. The clock counter gets the value x if $s.\mathit{clock\text{-}counter}_c = x - 1$. From Invariant C.26 we know that $\mathit{first(tick_c)} - \mathit{clock\text{-}rate} \geq \mathit{lst\text{-}time\text{-}cc}_c(x - 1)$, and by the inductive hypothesis we know that in state s, $s.\mathit{lst\text{-}time\text{-}cc}(x - 1) - s.\mathit{lst\text{-}time\text{-}cc}(x - j - 1) \geq j \times \mathit{clock\text{-}rate}$. Since this step does not change the value of $\mathit{lst\text{-}time\text{-}cc}(x - j - 1)$, we know that after this step $\mathit{lst\text{-}time\text{-}cc}_c(x) - \mathit{lst\text{-}time\text{-}cc}_c(x - j - 1) \geq (j + 1) \times \mathit{clock\text{-}rate}$. Thus, Part 1 holds for $j + 1$.

2. The proof for Part 2 is symmetric. ■

Invariant C.30

1. If $\mathit{clock\text{-}counter}_c = x \ \wedge \ \mathit{lst\text{-}time\text{-}cc}_c(x) \neq 0 \ \wedge \ i \in \mathtt{BN}$ then $\mathit{lst\text{-}time\text{-}cc}_c(x) - \mathit{lst\text{-}time\text{-}cc}_c(x - i) \geq \min(qt, i \times \mathit{clock\text{-}rate})$.

2. If $\mathit{clock\text{-}counter}_s = x \ \wedge \ \mathit{lst\text{-}time\text{-}cc}_s(x) \neq 0 \ \wedge \ i \in \mathtt{BN}$ then $\mathit{lst\text{-}time\text{-}cc}_s(x) - \mathit{lst\text{-}time\text{-}cc}_s(x - i) \geq \min(qt, i \times \mathit{clock\text{-}rate})$.

Proof: For this invariant we have two levels of induction. The first level is induction on i, and the second level is the induction on the steps of $\mathcal{BTCP}^h$. The base case of the induction on i is for $i = 0$. The invariant clearly holds for this case. For the inductive case we assume that the Invariant holds for j and show it holds for $j + 1$. We show it holds for

$j+1$ by induction on the steps of $\mathcal{BTCP}^h$. In the start state for all x *lst-time-cc*$_c(x)$ and *lst-time-cc*$_s(x)$ have the value 0, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. <u>a = *clock-counter-tick*$_c$</u>

   This step may cause the premise of the Part 1 to go from false to true by assigning *clock-counter*$_c$ to the value x and *lst-time-cc*$_c(x)$ to *now*. This step is enabled if *now* $\geq$ *first(tick*$_c$*)*. The clock counter gets the value x if s.*clock-counter*$_c = x-1$. From Invariant C.26 we know that *first(tick*$_c$*)* $-$ *clock-rate* $\geq$ *lst-time-cc*$_c(x-1)$, and by the inductive hypothesis we know that in state s, s.*lst-time-cc*$(x-1) - s$.*lst-time-cc*$(x-j-1) \geq \min(qt, j \times$ *clock-rate*$)$. Since this step does not change the value of *lst-time-cc*$(x-j-1)$, we know that after this step *lst-time-cc*$_c(x) -$ *lst-time-cc*$_c(x-j-1) \geq \min(qt, (j+1) \times$ *clock-rate*$)$. Thus, Part 1 holds for $j+1$.

   <u>a = *recover*$_c$</u>

   This step may cause the premise of the Part 1 to go from false to true by assigning *clock-counter*$_c$ to the value x and *lst-time-cc*$_c(x)$ to *now*. This step is enabled if *now* $\geq$ *first(recov*$_c$*)*. From Invariant C.25 we know that in state s *lst-time-cc*$_c(x-j-1) \leq$ *first(recov*$_c$*)* $- qt$. Since this step does not change *lst-time-cc*$_c(x-j-1)$, we know Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant C.31

1. If *clock-counter*$_c = x$ then *lst-time-cc*$_c(x)$ + *clock-rate* = *last(tick*$_c$*)*.

2. If *clock-counter*$_s = x$ then *lst-time-cc*$_s(x)$ + *clock-rate* = *last(tick*$_s$*)*.

Proof: In the start state, for all x, *lst-time-cc*$_c(x)$ and *lst-time-cc*$_s(x)$ have the value 0, and *last(tick*$_c$*)* and *last(tick*$_s$*)* have the value *clock-rate*, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. <u>a = *clock-counter-tick*$_c$ and a = *recover*$_c$</u>

   These steps may cause the premise of the Part 1 to go from false to true by assigning

$clock\text{-}counter_c$ to the value x and $lst\text{-}time\text{-}cc_c(x)$ to now. In these steps $last(tick_c)$ is also assigned $now + clock\text{-}rate$, so Part 1 holds.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant C.32

1. If $mode_c \neq \texttt{rec} \;\wedge\; clock\text{-}counter_c = x$ then $lst\text{-}time\text{-}cc_c(x) \geq now - clock\text{-}rate$.

2. If $mode_s \neq \texttt{rec} \;\wedge\; clock\text{-}counter_s = x$ then $lst\text{-}time\text{-}cc_s(x) \geq now - clock\text{-}rate$.

Proof: In the start state both $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = clock\text{-}counter\text{-}tick_c}$

   This step may cause the premise of the Part 1 to go from false to true by assigning $clock\text{-}counter_c$ to the value x and $lst\text{-}time\text{-}cc_c(x)$ to now. Clearly Part 1 holds after this step.

   $\underline{a = \nu(t)}$

   This step may cause the consequence of Part 1 to go from true to false, if $s.now + t > s.lst\text{-}time\text{-}cc_c(x) + clock\text{-}rate$. If $mode_c = \texttt{rec}$ then the premise of Part 1 is also false. If $mode_c \neq \texttt{rec}$ then this step is only enabled if $s.now + t \leq last(tick_c)$. From Invariant C.31 we know that if $clock\text{-}counter_c = x$ then $lst\text{-}time\text{-}cc_c(x) + clock\text{-}rate = last(tick_c)$. Therefore, if this step is enabled $clock\text{-}counter_c \neq x \vee mode_c = \texttt{rec}$, and since this step does not change $clock\text{-}counter_c$ or $mode_c$, Part 1 holds.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant C.33

1. If $(p, t) \in in\text{-}transit_{cs}$ then $lst\text{-}time\text{-}cc_c(sn(p)) \leq t \;\wedge\; lst\text{-}time\text{-}sn_c(sn(p)) \leq t$.

2. If $(p, t) \in in\text{-}transit_{sc}$ then $lst\text{-}time\text{-}cc_s(sn(p)) \leq t \wedge\; lst\text{-}time\text{-}sn_s(sn(p)) \leq t$.

In the start state, both $in\text{-}transit_{cs}$ and $in\text{-}transit_{sc}$ are empty, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = send\text{-}seg_{cs}(p)}$

   These steps add segments to $in\text{-}transit_{cs}$. The timestamp t on any of these segments is $now + \mu$. From Invariant C.2 we know that for all $x \in \texttt{BN}$, $\mathit{lst\text{-}time\text{-}cc}_c(x) \leq now$ and $\mathit{lst\text{-}time\text{-}sn}_c(x) \leq now$. Therefore, Part 1 holds after these steps.

$\underline{a = \mathit{clock\text{-}counter\text{-}tick}_c \text{ and } a = \mathit{recover}_c}$

These steps may assign $\mathit{lst\text{-}time\text{-}cc}_c(sn(p))$ to now, and so can cause the consequence of Part 1 to go from true to false, if $now > t$. However, from Invariant C.1 we know that for any $(p, t) \in \mathit{in\text{-}transit}_{cs}$, $now \leq t$. Therefore, the premise must be false also, so Part 1 holds after these steps.

$\underline{a = \mathit{send\text{-}msg}_c(\mathit{open},\ m,\ \mathit{close}) \text{ and } a = \mathit{prepare\text{-}msg}_c}$

These steps may assign $\mathit{lst\text{-}time\text{-}sn}_c(sn(p))$ to now, and so can cause the consequence of Part 1 to go from true to false, if $now > t$. However, from Invariant C.1 we know that for any $(p, t) \in \mathit{in\text{-}transit}_{cs}$, $now \leq t$. Therefore, the premise must be false also, so Part 1 holds after these steps.

2. The proof for Part 2 is symmetric. ■

Invariant C.34

1. If $mode_c = \texttt{syn-sent} \wedge \mathit{wait\text{-}t\_o}_c = \infty$ then $\mathit{lst\text{-}time\text{-}cc}_c(sn_c) \geq \mathit{last(response_c)} - rt - \mathit{clock\text{-}rate}$.

2. If $mode_s = \texttt{syn-rcvd} \wedge \mathit{wait\text{-}t\_o}_s = \infty$ then $\mathit{lst\text{-}time\text{-}cc}_s(sn_s) \geq \mathit{last(response_s)} - rt - \mathit{clock\text{-}rate}$.

Proof: In the start state both $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below. For this invariant the steps with $a = \mathit{clock\text{-}counter\text{-}tick}_c$ and $a = \mathit{recover}_c$ for Part 1 and the steps with $a = \mathit{clock\text{-}counter\text{-}tick}_s$ and $a = \mathit{recover}_s$ for Part 2 are not critical even though they assign the value of now to $\mathit{lst\text{-}time\text{-}cc}_c(\mathit{clock\text{-}counter}_c)$ and $\mathit{lst\text{-}time\text{-}cc}_s(\mathit{clock\text{-}counter}_s)$ respectively. They are not critical because from Invariant C.2 we know that for all $x \in \texttt{BN}$, $now \geq \mathit{lst\text{-}time\text{-}cc}_c(x)$ and $now \geq \mathit{lst\text{-}time\text{-}cc}_s(x)$. Thus, because these steps can only cause $\mathit{lst\text{-}time\text{-}cc}_c(x)$ and $\mathit{lst\text{-}time\text{-}cc}_s(x)$ to increase, they steps cannot cause the consequence of either part to go from true to false.

1. $\underline{a = \mathit{send\text{-}msg}_c(\mathit{open},\ m,\ \mathit{close})}$

   This step can make the premise of Part 1 to go from false to true. From Invariant C.32

we know that the consequence is also true after this step.

$a = \textit{send-seg}_{cs}(p)$

These steps may cause the consequence of Part 1 to go from true to false. In the case where $p = (SYN, sn_c)$, after that step $\textit{wait-t\_o}_c \neq \infty$. After the other steps $\textit{mode}_c \neq$ `syn-sent`, so Part 1 holds.

$a = \textit{receive-seg}_{sc}(p)$

These steps may also cause the consequence on Part 1 to go from true to false. However, after these steps, $\textit{mode}_c \neq$ `syn-sent`, so Part 1 holds.

2. The proof for Part 2 is symmetric. ■

Invariant C.35

1. If $\textit{mode}_c =$ `syn-sent` $\wedge\ \textit{wait-t\_o}_c = \infty$ then $\textit{lst-time-cc}_c(sn_c) \geq \textit{now} - \textit{rt} - \textit{clock-rate}$.

2. If $\textit{mode}_s =$ `syn-rcvd` $\wedge\ \textit{wait-t\_o}_s = \infty$ then $\textit{lst-time-cc}_s(sn_s) \geq \textit{now} - \textit{rt} - \textit{clock-rate}$.

Proof: In the start state both $\textit{mode}_c$ and $\textit{mode}_s$ have the value `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $a = \textit{send-msg}_c(\textit{open}, m, \textit{close})$

   This step can make the premise of Part 1 to go from false to true. From Invariant C.32 we know that the consequence is also true after this step.

   $a = \nu(t)$

   This step may cause the consequence of Part 1 to go from true to false. It is only enabled if $s.\textit{now} + t \leq \textit{last}(\textit{response}_c)$. From Invariant C.34 we know that if $\textit{mode}_c =$ `syn-sent` $\wedge\ \textit{wait-t\_o}_c = \infty$ then $\textit{lst-time-cc}_c(sn_c) \geq \textit{last}(\textit{response}_c) - \textit{rt} - \textit{clock-rate}$. Therefore, if this step causes the consequence of Part 1 to be false, then the premise must also be false.

2. The proof for Part 2 is symmetric. ■

Invariant C.36

1. If $\textit{mode}_c =$ `syn-sent` $\wedge\ \textit{wait-t\_o}_c \neq \infty\ \wedge\ \textit{now} \leq \textit{wait-t\_o}_c$ then $\textit{lst-time-cc}_c(sn_c) \geq \textit{wait-t\_o}_c - \textit{wt} - \textit{rt} - \textit{clock-rate}$.

2. If $mode_s = \texttt{syn-rcvd} \ \wedge \ wait\text{-}t\_o_s \neq \infty \ \wedge \ now \leq wait\text{-}t\_o_s$ then $lst\text{-}time\text{-}cc_s(sn_s) \geq wait\text{-}t\_o_s - wt - rt - clock\text{-}rate$.

Proof: In the start state both $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = send\text{-}seg_{cs}(SYN,\ sn_c)}$

   This step can make the premise of Part 1 to go from false to true by assigning $wait\text{-}t\_o_c$ to $now + wt$, if $s.mode_c = \texttt{syn-sent} \wedge s.wait\text{-}t\_o_c = \infty$. However, from Invariant C.35 we know that $s.lst\text{-}time\text{-}cc_c(sn_c) \geq s.now - rt - clock\text{-}rate$. Therefore, since this step does not change $lst\text{-}time\text{-}cc_c(sn_c)$ or now, Part 1 holds after this step.

   $\underline{a = receive\text{-}seg_{sc}(p)}$

   These steps may cause the consequence of Part 1 to go from true to fales by assigning $wait\text{-}t\_o_c$ to ∞. However, after each of these steps the premise is also clearly false, so Part 1 holds.

2. The proof for Part 2 is symmetric. ■

Invariant C.37

1. If $mode_c = \texttt{syn-sent} \ \wedge \ now \leq wait\text{-}t\_o_c$ then $lst\text{-}time\text{-}cc_c(sn_c) \geq now - wt - rt - clock\text{-}rate$.

2. If $mode_s = \texttt{syn-rcvd} \ \wedge \ now \leq wait\text{-}t\_o_s$ then $lst\text{-}time\text{-}cc_s(sn_s) \geq now - wt - rt - clock\text{-}rate$.

Proof: In the start state both $mode_c$ and $mode_s$ have the value `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = send\text{-}msg_c(open,\ m,\ close)}$

   This step can make the premise of Part 1 to go from false to true. In this step sn_c is assigned to $clock\text{-}counter_c$. From Invariant C.32 we know that $lst\text{-}time\text{-}cc_c(sn_c) + clock\text{-}rate \geq now$, so Part 1 holds after this step.

   $\underline{a = \nu(t)}$

   This step may cause the consequence of Part 1 to go from true to false, if $s.now +$

$t > s.lst\text{-}time\text{-}cc_c(sn_c) + wt + rt + clock\text{-}rate$. From Invariant C.36 we know that if $mode_c = \texttt{syn-sent} \wedge wait\text{-}t\_o_c \neq \infty \wedge now \leq wait\text{-}t\_o_c$ then $lst\text{-}time\text{-}cc_c(sn_c) \geq wait\text{-}t\_o_c - wt - rt - clock\text{-}rate$, and from Invariant C.35 we know that if $mode_c = \texttt{syn-sent} \wedge wait\text{-}t\_o_c = \infty$ then $lst\text{-}time\text{-}cc_c(sn_c) \geq now - rt - clock\text{-}rate$. Therefore, if $s'.now > s'.lst\text{-}time\text{-}cc_c(sn_c) + wt + rt + clock\text{-}rate$, then $s'.now > wait\text{-}t\_o_c$, so the premise of Part 1 is also false.

2. $\underline{a = receive\text{-}seg_{cs}(SYN,\ sn_c)}$

   This step can make the premise of Part 2 to go from false to true. In this step sn_s is assigned to $clock\text{-}counter_s$. From Invariant C.32 we know that $lst\text{-}time\text{-}cc_s(sn_s) + clock\text{-}rate \geq now$, so Part 2 holds after this step.

   $\underline{a = \nu(t)}$

   The proof for this step is symmetric to the proof for the same step for Part 1. ■

Invariant C.38

1. If there exists SYN segment $(p, t) \in in\text{-}transit_{cs}$ then $t \leq lst\text{-}time\text{-}cc_c(sn(p)) + clock\text{-}rate + wt + rt + \mu$.

2. If there exists a SYN segment $(p, t) \in in\text{-}transit_{sc}$ then $t \leq lst\text{-}time\text{-}cc_s(sn(p)) + clock\text{-}rate + wt + rt + \mu$.

In the start state, both $in\text{-}transit_{cs}$ and $in\text{-}transit_{sc}$ are empty, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. $\underline{a = send\text{-}seg_{cs}(SYN,\ sn_c)}$

   This step adds a SYN segment to $in\text{-}transit_{cs}$ if $mode_c = \texttt{syn-sent} \wedge now \leq wait\text{-}t\_o_c$. The timestamp on this segment is $now + \mu$. From Invariant C.37 we know that if $mode_c = \texttt{syn-sent} \wedge now \leq wait\text{-}t\_o_c$ then $lst\text{-}time\text{-}cc_c(sn_c) \geq now - wt - rt - clock\text{-}rate$. Therfore, after this step Part 1 holds.

2. The proof for Part 2 is symmetric. ■

Invariant C.39

1. If there exists a SYN segment $(p, t) \in in\text{-}transit_{cs}$ then $clock\text{-}counter_c \geq sn(p)$.

2. If there exists a SYN segment $(p,t) \in \textit{in-transit}_{sc}$ then $\textit{clock-counter}_s \geq \textit{sn}(p)$.

Proof: In the start state $\textit{in-transit}_{cs}$ and $\textit{in-transit}_{sc}$ are both empty, so the invariant holds for this state. We consider critical steps of the form (s, a, s') below.

1. $\underline{a = \textit{send-seg}_{cs}(\textit{SYN}, \textit{sn}_c)}$

   This step adds a SYN segment to (p,t) to $\textit{in-transit}_{cs}$, where $t = \textit{now} + \mu$. After this segment is added, we know that if $\textit{sn}(p) > \textit{clock-counter}_c$ then by Invariant C.30, $\textit{lst-time-cc}_c(\textit{clock-counter}_c) - \textit{lst-time-cc}_c(\textit{sn}(p)) \geq \textit{qt}$. By Invariant C.32 we know $\textit{lst-time-cc}_c(\textit{clock-counter}_c) \geq \textit{now} - \textit{clock-rate}$. Therefore, $\textit{lst-time-cc}_c(\textit{sn}(p)) \leq \textit{now} - \textit{clock-rate} - \textit{qt}$. From Invariant C.38 we know that $t \leq \textit{lst-time-cc}_c(\textit{sn}(p)) + \textit{clock-rate} + \textit{wt} + \textit{rt} + \mu$. Combining the two inequalities gives us, $t \leq \textit{now} - \textit{clock-rate} - \textit{qt} + \textit{clock-rate} + \textit{wt} + \textit{rt} + \mu$. Thus, $t \leq \textit{now} - \textit{qt} + \textit{wt} + \textit{rt} + \mu$. Since $\textit{qt} > \mu + \textit{wt} + \textit{rt}$, we get $t < \textit{now}$. However, this contradicts the fact that $t = \textit{now} + \mu$. Therefore, $\textit{sn}(p) \leq \textit{clock-counter}_c$, so Part 1 holds after this step.

   $\underline{a = \textit{recover}_c}$

   This step may cause the consequence of Part 1 to go from true to false by assigning $\textit{clock-counter}_c$ an arbitrary value. If is enabled if $\textit{mode}_c = \texttt{rec} \wedge \textit{now} \geq \textit{first}(\textit{recov}_c)$. Invariant C.15 tells us that when this step is enabled that $\textit{in-transit}_{cs}$ is empty.

   $\underline{a = \textit{clock-counter-tick}_c}$

   The step may also cause the consequence of Part 1 to go from true to false by incrementing $\textit{clock-counter}_c$. However, as we showed for the case of the step with $a = \textit{send-seg}_{cs}(\textit{SYN}, \textit{sn}_c)$, there is can be no SYN segment on $\textit{in-transit}_{cs}$ with $\textit{sn}(p) > \textit{clock-counter}_c$ if this is the case.

2. The proof for Part 2 is symmetric. ■

Invariant C.40

If $\textit{mode}_c = \texttt{closed} \wedge \textit{last}(\textit{tick}_c) > \textit{first}(\textit{open}_c)$ then $\textit{clock-counter}_c \neq \textit{sn}(p)$ for any SYN segment $(p,t) \in \textit{in-transit}_{cs}$.

Proof: In the start state $\textit{clock-counter}_c = \textit{clock-counter}_s = 0$ and $\textit{in-transit}_{cs}$ and $\textit{in-transit}_{sc}$ are both empty so the consequence of both parts of the invariant is true, so it holds in the start state. We consider critical steps of the form (s, a, s') below.

$a = send\text{-}seg_{cs}(SYN,\ sn_c)$

This step adds a SYN segment to $in\text{-}transit_{cs}$, but only if $mode_c = \texttt{syn-sent}$.

$a = clock\text{-}counter\text{-}tick_c$

This step may cause the premise of the invariant to go from false to true by assigning $last(tick_c)$ to $now + clock\text{-}rate$. This step also increments $clock\text{-}counter_c$. Since we know from Invariant C.39 that $s.clock\text{-}counter_c \geq sn(p)$ for any SYN segment $(p, t) \in in\text{-}transit_{cs}$, we know $s'.clock\text{-}counter_c \neq sn(p)$.

$a = recover_c$

This step may also cause the premise of the invariant to go from false to true by assigning $last(tick_c)$ to $now + clock\text{-}rate$. This step is enabled if $s.mode_c = \texttt{rec} \wedge now \geq first(recov_c)$. However, Invariant C.15 tells us that when this step is enabled there are no segments in $in\text{-}transit_{cs}$. ■

Invariant C.41

$mode_s = \texttt{syn-rcvd} \wedge wait\text{-}t\_o_s = \infty$ then $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq last(response_s) - wt - \mu - 2rt - clock\text{-}rate$.

Proof: In the start state $mode_s = \texttt{closed}$, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$a = receive\text{-}seg_{cs}(SYN,\ sn_c)$

This step may cause the premise of the invariant to go from false to true. Let (p, t) be the segment received. This step assigns $last(response_s)$ to $now + rt$, and ack_s to $sn(p) + 1$. From Invariant C.38 we know $t \leq lst\text{-}time\text{-}cc_c(sn(p)) + clock\text{-}rate + wt + rt + \mu$, and from Invariant C.1 we know $now \leq t \leq now + \mu$. Thus, $now \leq lst\text{-}time\text{-}cc_c(sn(p)) + clock\text{-}rate + wt + rt + \mu$. Since $s'.last(response_s) = s.now + rt$, we know the invariant holds after this step.

$a = send\text{-}seg_{sc}(p)$

These steps may cause the consequence on the invariant to go from true to false. In the case where $p = (SYN, sn_s, ack_s)$, after that step $wait\text{-}t\_o_s \neq \infty$. After the other steps $mode_s \neq \texttt{syn-rcvd}$, so the invariant holds.

$a = receive\text{-}seg_{sc}(p)$

These steps may also cause the consequence on Part 1 to go from true to false. However, after these steps, $mode_s \neq$ `syn-rcvd`, so the invariant holds. ■

Invariant C.42

If $mode_s =$ `syn-rcvd` $\wedge$ $wait\text{-}t\_o_s = \infty$ then $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq now - wt - \mu - 2rt - clock\text{-}rate$.

Proof: In the start state $mode_s =$ `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$a = receive\text{-}seg_{cs}(SYN,\ sn_c)$

This step may cause the premise of the invariant to go from false to true. Let (p, t) be the segment received. From Invariant C.38 we know $t \leq lst\text{-}time\text{-}cc_c(sn(p)) + clock\text{-}rate + wt + rt + \mu$, and from Invariant C.1 we know $now \leq t$. Therefore, the invariant holds after this step.

$a = \nu(t)$

This step may cause the consequence of Part 1 to go from true to false. It is only enabled if $s.now + t \leq last(response_s)$. From Invariant C.41 we know that $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq last(response_s) - wt - \mu - 2rt - clock\text{-}rate$. Therefore, if this step causes the consequence of the invariant to be false, then the premise must also be false. ■

Invariant C.43

If $mode_s =$ `syn-rcvd` $\wedge$ $wait\text{-}t\_o_s \neq \infty$ $\wedge$ $now \leq wait\text{-}t\_o_s$ then $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq wait\text{-}t\_o_s - \mu - 2wt - 2rt - clock\text{-}rate$.

Proof: In the start state $mode_s =$ `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$a = send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$

This step can make the premise of Part 1 to go from false to true by assigning $wait\text{-}t\_o_s$ to $now + wt$, if $s.mode_s =$ `syn-rcvd` $\wedge$ $s.wait\text{-}t\_o_s = \infty$. However, from Invariant C.42 we know that $s.lst\text{-}time\text{-}cc_c(ack_s - 1) \geq s.now - wt - \mu - 2rt - clock\text{-}rate$. Therefore, since this step does not change $lst\text{-}time\text{-}cc_c(ack_s - 1)$ or now, the invariant holds after this step.

$a = receive\text{-}seg_{cs}(p)$

These steps may cause the consequence of the invariant to go from true to fales by assigning *wait-t_o*$_s$ to ∞. However, after each of these steps the premise also clearly false, so the invariant holds. ■

Invariant C.44

If $mode_s$ = `syn-rcvd` $\wedge$ $now \leq wait\text{-}t\_o_s$ then $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq now - \mu - 2wt - 2rt - clock\text{-}rate$.

Proof: In the start state $mode_s$ = `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

<u>a = *receive-seg*$_{cs}$(*SYN*, sn_c)</u>

This step may cause the premise of the invariant to go from false to true. Let (p, t) be the segment received. From Invariant C.38 we know $t \leq lst\text{-}time\text{-}cc_c(sn(p)) + clock\text{-}rate + wt + rt + \mu$, and from Invariant C.1 we know $now \leq t$. Therefore, the invariant holds after this step.

<u>$a = \nu(t)$</u>

This step may cause the consequence of Part 1 to go from true to false, if $s.now + t > s.lst\text{-}time\text{-}cc_c(ack_s - 1) + \mu + 2wt + 2rt + clock\text{-}rate$. From Invariant C.43 we know that if $mode_s$ = `syn-rcvd` $\wedge$ $wait\text{-}t\_o_s \neq \infty \wedge now \leq wait\text{-}t\_o_s$ then $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq wait\text{-}t\_o_s - \mu - 2wt - 2rt - clock\text{-}rate$, and from Invariant C.42 we know that if $mode_s$ = `syn-rcvd` $\wedge$ $wait\text{-}t\_o_s = \infty$ then $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq now - \mu - wt - 2rt - clock\text{-}rate$. Therefore, if $s'.now > s'.lst\text{-}time\text{-}cc_c(ack_s - 1) + \mu + 2wt + 2rt + clock\text{-}rate$, then $s'.now > wait\text{-}t\_o_s$, so the premise of the invariant is also false. ■

Invariant C.45

If there exists a SYN segment $(p, t) \in in\text{-}transit_{sc}$ then $t \leq lst\text{-}time\text{-}cc_c(ack(p) - 1) + clock\text{-}rate + 2(wt + rt + \mu)$.

Proof: In the start state $in\text{-}transit_{sc}$ is empty, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

<u>a = *send-seg*$_{sc}$(*SYN*, sn_s, ack_s)</u>

This step adds a SYN segment to $in\text{-}transit_{sc}$ if $mode_s$ = `syn-rcvd` $\wedge$ $now \leq wait\text{-}t\_o_s$. The timestamp on this segment is $now + \mu$. From Invariant C.44 we know that if $mode_s$ =

syn-rcvd $\wedge$ $now \leq wait\text{-}t\_o_s$ then $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq now - \mu - 2wt - 2rt - clock\text{-}rate$. Therfore, after this step the invariant holds. ∎

Invariant C.46

If $mode_s =$ `syn-rcvd` $\wedge$ $now \leq wait\text{-}t\_o_s$ then $clock\text{-}counter_c \geq ack_s - 1$.

Proof: In the start state $mode_s =$ `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

<u>a = *receive-seg*$_{cs}$*(SYN,* sn_c*)*</u>

This step may cause the premise of the invariant to go from false to true. Let (p, t) be the segment received. This step also assigns ack_s to $sn(p) + 1$. Therefore, if the consequence of the invariant is not also true, then it must be that $sn(p) > clock\text{-}counter_c$. We know show by contradiction that there cannot be a SYN segment with $sn(p) > clock\text{-}counter_c$. If $sn(p) > clock\text{-}counter_c$, then by Invariant C.30, $lst\text{-}time\text{-}cc_c(clock\text{-}counter_c) - lst\text{-}time\text{-}cc_c(sn(p)) \geq qt$. By Invariant C.32 we know $lst\text{-}time\text{-}cc_c(clock\text{-}counter_c) \geq now - clock\text{-}rate$. Therefore, $lst\text{-}time\text{-}cc_c(sn(p)) \leq now - clock\text{-}rate - qt$. From Invariant C.38 we know that $t \leq lst\text{-}time\text{-}cc_c(sn(p)) + clock\text{-}rate + wt + rt + \mu$. Combining the two inequalities we get $t \leq now - clock\text{-}rate - qt + clock\text{-}rate + wt + rt + \mu$. Thus, $t \leq now - qt + wt + rt + \mu$. Since $qt > \mu + wt + rt$, we get $t < now$. However, this contradicts Invariant C.1 which says that for all $(p, t) \in in\text{-}transit_{cs}$, $now \leq t$. Thus, there cannot be a SYN segment in $in\text{-}transit_{cs}$ with $sn(p) > clock\text{-}counter_c$. Therefore, the invariant holds after this step.

<u>a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*</u>

These steps may cause the consequence of Part 1 to go from true to false by changing ack_s. However, the change happens only if $mode_s \neq$ `syn-rcvd`.

<u>$a = recover_c$</u>

This step may cause the consequence of the invariant to go from true to false by assigning $clock\text{-}counter_c$ an arbitrary value. This step is enabled if $s.mode_c =$ `rec` $\wedge$ $now \geq$ *first(recov*$_c$*)*. However, from Invariant C.13 we know that if $mode_c =$ `rec` $\wedge$ $mode_s =$ `syn-rcvd` $\wedge$ $now \leq wait\text{-}t\_o_s$ then *first(recov*$_c$*)* $> now + \mu$. Therefore, if the step causes the consequence of the invariant to be false, the premise must also be false.

<u>$a = clock\text{-}counter\text{-}tick_c$</u>

The step may also cause the consequence of the invariant to go from true to false by incrementing $clock\text{-}counter_c$, so that $s'.clock\text{-}counter_c < s'.ack_s - 1$. This step also sets $lst\text{-}time\text{-}cc_c(s'.clock\text{-}counter_c)$ to now. If $s'.clock\text{-}counter_c < s'.ack_s - 1 \wedge lst\text{-}crash\text{-}time_c < lst\text{-}time\text{-}cc_c(s'.ack_s - 1)$, then by Invariant C.29 $lst\text{-}time\text{-}cc_c(s'.clock\text{-}counter_c) - lst\text{-}time\text{-}cc_c(s'.ack_s - 1) \geq ct/2$. Since $lst\text{-}time\text{-}cc_c(s'.clock\text{-}counter_c) = now$, we get $now - lst\text{-}time\text{-}cc_c(s'.ack_s - 1) \geq ct/c$. From Invariant C.44 we also know that $lst\text{-}time\text{-}cc_c(ack_s - 1) \geq now - \mu - 2wt - 2rt - clock\text{-}rate$. Since $ct/2 > \mu + 2wt + 2rt + clock\text{-}rate$, we get a contradiction for the case where $lst\text{-}crash\text{-}time_c < lst\text{-}time\text{-}cc_c(s'.ack_s - 1)$. Thus, the premise of the invariant must also be false in this situation. If $lst\text{-}crash\text{-}time_c \geq ltime_c(s'.ack_s - 1)$, then by Invariant C.20 $\neg(mode_s = \texttt{syn-rcvd} \wedge now \leq wait\text{-}t\_o_s)$, so the invariant holds after this step. ■

Invariant C.47

If there exists a SYN segment $(p, t) \in in\text{-}transit_{sc}$ then $clock\text{-}counter_c \geq ack(p) - 1$.

In the start state $in\text{-}transit_{cs}$ is empty, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

$\underline{a = send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)}$

This step adds a SYN segment to $in\text{-}transit_{sc}$ if $mode_s = \texttt{syn-rcvd} \wedge now \leq wait\text{-}t\_o_s$. By Invariant C.46 we know this invariant holds after this step.

$\underline{a = recover_c}$

This step may cause the consequence of Part 1 to go from true to false by assigning $clock\text{-}counter_c$ an arbitrary value. However, Invariant C.15 tells us that when this step is enabled there are no SYN segments in $in\text{-}transit_{sc}$.

$\underline{a = clock\text{-}counter\text{-}tick_c}$

The step may also cause the consequence of Part 1 to go from true to false by incrementing $clock\text{-}counter_c$, so that $s'.clock\text{-}counter_c < ack(p) - 1$ for a SYN segment $(p, t) \in in\text{-}transit_{sc}$. This step also sets $lst\text{-}time\text{-}cc_c(s'.clock\text{-}counter_c)$ to now. If $s'.clock\text{-}counter_c < ack(p) - 1 \wedge lst\text{-}crash\text{-}time_c < lst\text{-}time\text{-}cc_c(ack(p) - 1)$, then by Invariant C.29 $lst\text{-}time\text{-}cc_c(s'.clock\text{-}counter_c) - lst\text{-}time\text{-}cc_c(ack(p) - 1) \geq ct/2$. Since $lst\text{-}time\text{-}cc_c(s'.clock\text{-}counter_c) = now$, we have $lst\text{-}time\text{-}cc_c(ack(p) - 1) \leq now - ct/2$. From Invariant C.45 we also know that $t \leq lst\text{-}time\text{-}cc_c(ack(p) - 1) + clock\text{-}rate + 2(\mu + wt + rt)$. Combining the two inequalities we get $t \leq now - ct/2 + clock\text{-}rate + 2(\mu + wt + rt)$. Since $ct/2 > 2(\mu + wt + rt) + clock\text{-}rate$, we get $t < now$. However, this contradicts Inariant C.1 which says for all segments in *in-transit*$_{cs}$ $t \leq now$. Therefore, we get a contradiction for the case where *lst-crash-time*$_c$ $<$ *lst-time-cc*$_c(ack(p) - 1)$. Thus, the premise of the invariant must also be false in this situation. If *lst-crash-time*$_c \geq ltime_c(ack(p) - 1)$, then by Invariant C.21 we know there are no SYN segments in *in-transit*$_{sc}$ with $ack(p) >$ *clock-counter*$_c$. ∎

Invariant C.48

If $mode_c = \texttt{closed} \wedge$ *last(tick*$_c$*)* $>$ *first(open*$_c$*)* then *clock-counter*$_c \neq ack_s - 1 \vee \neg(mode_s = \texttt{syn-rcvd} \wedge now \leq$ *wait-t_o*$_s$).

Proof: In the start state *clock-counter*$_c = 0$ and *in-transit*$_{sc}$ is empty so the consequence of the invariant is true, so it holds in the start state. We consider critical steps of the form (s, a, s') below.

$\underline{a = \textit{clock-counter-tick}_c}$

This step may cause the premise of the invariant to go from false to true by assigning *last(tick*$_c$*)* to $now +$ *clock-rate*. This step also increments *clock-counter*$_c$. Since by Invariant C.46 if $mode_s = \texttt{syn-rcvd} \wedge now \leq$ *wait-t_o*$_s$ then *clock-counter*$_c \geq ack_s - 1$, we know that *clock-counter*$_c \neq ack_s - 1 \vee \neq (mode_s = \texttt{syn-rcvd} \wedge now \leq$ *wait-t_o*$_s$).

$\underline{a = \textit{recover}_c}$

This step may also cause the premise of the invariant to go from false to true by assigning *last(tick*$_c$*)* to $now +$ *clock-rate*. This step is enabled if $s.mode_c = \texttt{rec} \wedge now \geq$ *first(recov*$_c$*)*. However, from Invariant C.13 we know that if $mode_c = \texttt{rec} \wedge mode_s = \texttt{syn-rcvd} \wedge now \leq$ *wait-t_o*$_s$ then *first(recov*$_c$*)* $> now + \mu$. Therefore, if the step causes the premise of the invariant to be true, the consequence must also be true.

$\underline{a = \textit{send-seg}_{cs}(\textit{SYN}, sn_c)}$

This step may cause the consequence of the invariant to go from true to false. This step sets ack_s to $sn(p) + 1$. However, form Invariant C.40 we know that if $mode_c = \texttt{closed} \wedge$ *last(tick*$_c$*)* $>$ *first(open*$_c$*)* there are no SYN segments in *in-transit*$_{cs}$ with $sn(p) =$ *clock-counter*$_c$. Therefore, if this step causes the consequence of the invariant to be false, the premise of the invariant must also be false. ∎

Invariant C.49

If $mode_c$ = closed $\wedge$ $last(tick_c) > first(open_c)$ then $clock\text{-}counter_c \neq ack(p) - 1$ for any SYN segment $(p, t) \in in\text{-}transit_{sc}$.

Proof: In the start state $clock\text{-}counter_c = 0$ and $in\text{-}transit_{sc}$ is empty so the consequence of the invariant is true, so it holds in the start state. We consider critical steps of the form (s, a, s') below.

<u>$a = clock\text{-}counter\text{-}tick_c$</u>

This step may cause the premise of the invariant to go from false to true by assigning $last(tick_c)$ to $now + clock\text{-}rate$. This step also increments $clock\text{-}counter_c$. Since we know from Invariant C.47 that $s.clock\text{-}counter_c \geq ack(p) - 1$ for any SYN segment $(p, t) \in in\text{-}transit_{sc}$, we know $s'.clock\text{-}counter_c \neq ack(p) - 1$.

<u>$a = recover_c$</u>

This step may also cause the premise of the invariant to go from false to true by assigning $last(tick_c)$ to $now + clock\text{-}rate$. This step is enabled if $s.mode_c$ = rec $\wedge$ $now \geq first(recov_c)$. However, Invariant C.15 tells us that when this step is enabled there are no SYN segments in $in\text{-}transit_{sc}$.

<u>$a = send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$</u>

This step adds a SYN segment to $in\text{-}transit_{sc}$, but only if $mode_s$ = syn-rcvd $\wedge$ $now \leq wait\text{-}t\_o_s$. From Invariant C.48 we know that if this step is enabled then $clock\text{-}counter_c \neq ack_s - 1$, so the invariant holds after this step. ■

Invariant C.50

1. If $mode_c$ = closed $\wedge$ $now > first(open_c)$ then $last(tick_c) > first(open_c)$.

2. If $mode_s \in \{$closed, listen$\}$ $\wedge$ $now > first(open_s)$ then $last(tick_s) > first(open_s)$.

Proof: . Follows from Invariant C.3. ■

Invariant C.51

1. If $mode_c$ = closed $\wedge$ $now > first(open_c)$ then $clock\text{-}counter_c \neq ack_s - 1 \vee \neg(mode_s$ = syn-rcvd $\wedge$ $now \leq wait\text{-}t\_o_s)$.

2. If $mode_c$ = closed $\wedge$ $now > first(open_c)$ then $clock\text{-}counter_c \neq ack(p) - 1$ for any SYN segment $(p, t) \in in\text{-}transit_{sc}$.

Proof: Part 1 follows from Invariants C.50 and C.48, and Part 2 follows from Invariants C.50 and C.49. ∎

Invariant 8.5

If $mode_c$ = `syn-sent` $\wedge$ $new\text{-}isn_c$ = `true` $\wedge$ $ack_s \in$ `BN` then $sn_c \geq ack_s \vee \neg(mode_s =$ `syn-rcvd` $\wedge\ now \leq wait\text{-}t\_o_s)$.

Proof: In the start state $mode_c$ = `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

<u>a = *send-msg*$_c$*(open, m, close)*</u>

This step can make the premise of the invariant to go from false to true if $s.mode_c$ = `closed` $\wedge$ $s.now > first(open_c)$. In this step sn_c is assigned to $s.clock\text{-}counter_c$. From Invariant C.51 we know that $clock\text{-}counter_c \neq ack_s - 1\ \vee \neg(mode_s =$ `syn-rcvd` $\wedge\ now \leq wait\text{-}t\_o_s)$, and from Invariant C.46 we know that if $mode_s$ = `syn-rcvd` $\wedge\ now \leq wait\text{-}t\_o_s$ then $clock\text{-}counter_c \geq ack_s - 1$. Therefore if $mode_s$ = `syn-rcvd` $\wedge\ now \leq wait\text{-}t\_o_s$ then $s.clock\text{-}counter_c > ack_s - 1$, so Part 1 holds after this step.

<u>a = *receive-seg*$_{cs}$*(SYN,* sn_c*)*</u>

This step that can make the consequence of the invariant go from true to false, but $new\text{-}isn_c$ is also set to `false`, so the invariant holds after this step.

<u>a = *prepare-msg*$_c$</u>

This step that can also make the consequence of the invariant go from true to false, but $mode_c \neq$ `syn-sent` after this step, so the invariant holds. ∎

Invariant 8.6

If $mode_c$ = `syn-sent` $\wedge$ $new\text{-}isn_c$ = `true` then for all SYN segments $(p, t) \in in\text{-}transit_{sc}$, $ack(p) < sn_c + 1$.

Proof: In the start state both $mode_c$ = `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

<u>a = *send-msg*$_c$*(open, m, close)*</u>

This step can make the premise of the invariant to go from false to true if $s.mode_c$ = `closed` $\wedge$ $s.now > first(open_c)$. In this step sn_c is assigned to $s.clock\text{-}counter_c$. From Invariant C.51 we know that for any SYN segment $(p, t) \in in\text{-}transit_{sc}$, $s.clock\text{-}counter_c \neq$

$ack(p)-1$. From Invariant C.47 we know s.$clock\text{-}counter_c \geq ack(p)-1$ for all SYN segments $(p,t) \in in\text{-}transit_{sc}$. Therefore, for all SYN segments $(p,t) \in in\text{-}transit_{sc}$, $ack(p) < sn_c + 1$, so the invariant holds after this step.

$\underline{a = send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)}$

This step that can make the consequence of the invariant go from true to false, by adding a SYN segment to $in\text{-}transit_{cs}$. However, this step is only enabled if $mode_c =$ `syn-rcvd` $\wedge$ $now \leq wait\text{-}t\_o_s$. From Invariant 8.5 we know that if this step is enabled, then the premise of the invariant must also be false, so the invariant holds after this step.

$\underline{a = prepare\text{-}msg_c}$

This step that can also make the consequence of the invariant go from true to false, but $mode_c \neq$ `syn-sent` after this step, so the invariant holds. ■

Invariant C.52

If $mode_c =$ `syn-sent` and there exists a SYN segment $(p,t) \in in\text{-}transit_{sc}$ with $ack(p) = sn_c + 1$, then $mode_s \neq$ `listen`.

Proof: If the start state $mode_s =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We know from Invariant 8.6 that the step with $a = send\text{-}msg_c(open,\ m,\ close)$ is not critical, because after this step there are no SYN segments $(p,t) \in in\text{-}transit_{sc}$ with $ack(p) = sn_c + 1$.

$\underline{a = passive\text{-}open}$

This step cause the consequence of the invariant to go from true to false by setting $mode_s$ to `listen` if $mode_s =$ `closed` $\wedge$ $now > first(open_s)$. However, we know from Invariants C.22 and C.1 that there are no segments in $in\text{-}transit_{sc}$, if $mode_s =$ `closed` $\wedge$ $now > first(open_s)$.

$\underline{a = send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)}$

This step may cause the premise of the invariant to go from false to true. However, this step is only enabled if $mode_s =$ `syn-rcvd`, so the invariant holds after this step. ■

Invariant C.53

If $lst\text{-}crash\text{-}time_s \geq lst\text{-}time\text{-}cc_s(sn(p))$ for SYN segment $(p,t) \in in\text{-}transit_{sc}$ then $t \leq lst\text{-}crash\text{-}time_s + \mu$.

Proof: In the start state $\mathit{in\text{-}transit}_{sc}$ is empty, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a = \mathit{crash}_s$

This step may cause the premise of the invariant to go from false to true. However, since $\mathit{lst\text{-}crash\text{-}time}_s$ is set to now, the invariant clearly holds after this step.

$a = \mathit{send\text{-}seg}_{sc}(\mathit{SYN},\ \mathit{sn}_s,\ \mathit{ack}_s)$

This step may cause the consequence of the invariant to go from true to false by adding a SYN segment to $\mathit{in\text{-}transit}_{sc}$ if $s.\mathit{mode}_s = \texttt{syn-rcvd} \wedge s.\mathit{now} \leq s.\mathit{wait\text{-}t\_o}_s$. However, it is easy to show that if $\mathit{mode}_s = \texttt{syn-rcvd}$ then $\mathit{lst\text{-}crash\text{-}time}_s < \mathit{lst\text{-}time\text{-}cc}_s(\mathit{sn}_s)$, so the premise is also false. ■

Invariant C.54

If $\mathit{ack\text{-}from\text{-}syn} = \texttt{true} \wedge \mathit{lst\text{-}crash\text{-}time}_s \geq \mathit{lst\text{-}time\text{-}cc}_s(\mathit{ack}_c - 1)$ and there exists a SYN segment $(p, t) \in \mathit{in\text{-}transit}_{cs}$ then $t \leq \mathit{lst\text{-}crash\text{-}time}_s + 2\mu$.

Proof: In the start state $\mathit{ack\text{-}from\text{-}syn}$ is undefined, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a = \mathit{receive\text{-}seg}_{sc}(\mathit{SYN},\ \mathit{sn}_s,\ \mathit{ack}_s)$

This step may cause the premise of the invariant to go from false to true. Let (q, t') be the SYN segment received in this step. From Invariant C.53 we know that $t' \leq \mathit{lst\text{-}crash\text{-}time}_s + \mu$, and from Invariant C.1 we know that $\mathit{now} \leq t'$ and $t \leq \mathit{now} + \mu$. Therefore, the invariant holds after this step.

$a = \mathit{send\text{-}seg}_{cs}(\mathit{SYN},\ \mathit{sn}_c)$

This step may cause the consequence of the invariant to go from true to false by adding a SYN segment to $\mathit{in\text{-}transit}_{cs}$. However, if this step is enabled $\mathit{ack\text{-}from\text{-}syn} = \texttt{false}$, so the invariant holds after this step. ■

Invariant C.55

If $\mathit{mode}_s = \texttt{listen} \wedge \mathit{ack\text{-}from\text{-}syn} = \texttt{true} \wedge \mathit{lst\text{-}crash\text{-}time}_s \geq \mathit{lst\text{-}time\text{-}cc}_s(\mathit{ack}_c - 1)$ then there are no SYN segments in $\mathit{in\text{-}transit}_{cs}$.

Proof: If the start state $\mathit{mode}_s = \texttt{closed}$, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. The step with $a = \mathit{receive\text{-}seg}_{sc}(\mathit{SYN},\ \mathit{sn}_s,\ \mathit{ack}_s)$

is not critical even though it may set *ack-from-syn* to `true`. It is not critical becasue we know from Invariant C.52 that $mode_s \neq$ `listen` if this step cause *ack-from-syn* to be `true`.

<u>a = *passive-open*</u>

This step cause the consequence of the invariant to go from true to false by setting $mode_s$ to `listen` if $mode_s =$ `closed` $\wedge$ $now > first(open_s)$. From Invariant C.17 we know that $lst\text{-}crash\text{-}time_s \leq now - qt$ and from Invariant C.54 we know that if there is a SYN segment $(p, t) \in in\text{-}transit_{cs}$ then $t \leq lst\text{-}crash\text{-}time_s + 2\mu$. Therefore we have $t \leq now - qt + 2\mu$. Since $qt > 2\mu$, there cannot be such a SYN segment, so the invariant holds after this step.

<u>$a = send\text{-}seg_{cs}(SYN,\ sn_c)$</u>

This step may cause the consequence of the invariant to go from true to false by adding a SYN segment to $in\text{-}transit_{cs}$. However, if this step is enabled *ack-from-syn* = `false`, so the invariant holds after this step. ∎

Invariant C.56

If $mode_s \in sync\text{-}states$ then $mode_c \neq$ `syn-sent` or there are no SYN segments $(p, t) \in in\text{-}transit_{sc}$ with $ack(p) = sn_c + 1$.

Proof: If the start state $mode_s =$ `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We know from Invariant 8.6 that the step with $a = send\text{-}msg_c(open,\ m,\ close)$ is not critical, because after this step there are no SYN segments $(p, t) \in in\text{-}transit_{sc}$ with $ack(p) = sn_c + 1$.

<u>$a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$ and $a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)$</u>

These steps may casue the premise of the invariant to go from false to true. However, since Invariant C.24 tells us that if $mode_c =$ `syn-sent` there are only SYN segments in $in\text{-}transit_{cs}$, the invariant holds after this step.

<u>$a = send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$</u>

This step may cause the consequence of the invariant to go from true to false. However, this step is only enabled if $mode_s =$ `syn-rcvd`. Therefore, the invariant holds after this step. ∎

Invariant C.57

If $mode_c =$ `syn-sent` and there exists a SYN segment $(p, t) \in in\text{-}transit_{sc}$ with $ack(p) = sn_c + 1$ then any segment $(p, t) \in in\text{-}transit_{sc}$ is a SYN segment.

Proof: If the start state $mode_s$ = `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. We know from Invariant 8.6 that the step with a = *send-msg*$_c$*(open, m, close)* is not critical, because after this step there are no SYN segments $(p, t) \in$ *in-transit*$_{sc}$ with $ack(p) = sn_c + 1$.

<u>a = *send-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*</u>

This step may cause the premise of the invariant to go from false to true. However, it is only enabled if $mode_s$ = `syn-rcvd` and from Invariant C.24, we know that any segment $(p, t) \in$ *in-transit*$_{sc}$ is a SYN segment.

<u>a = *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and a = *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)*</u>

These steps may cause the consequence of the invariant to go from true to false. These steps are enabled if $mode_s \in$ *sync-states*. However, from Invariant C.56 we know that if these steps are enabled, then the premise of the invariant is false. ∎

Invariant C.58

If $mode_c \in$ *sync-states* and there exists a SYN segment $(p, t) \in$ *in-transit*$_{cs}$ and a non-SYN segment $(q, t') \in$ *in-transit*$_{sc}$ then $t \leq t'$.

Proof: If the start state $mode_c$ = `closed`, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

<u>a = *receive-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*</u>

This step may cause the premise of the invariant to go from false to true if $s.mode_c$ = `syn-sent` $\wedge [ack_s] = s.sn_c + 1$. However, from Invariant C.57 we know any other segments in *in-transit*$_{sc}$ are SYN segments. Thus, the invariant holds after this step.

<u>a = *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and a = *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)*</u>

These steps may cause the premise of the invariant to go from true to false. The timestamp on a segment sent by either of these actions is $now + \mu$. From Invariant C.1 we know that $t \leq now + \mu$, for any segment $(p, t) \in transit_{cs}$. Therefore, the invariant holds after these steps. ∎

Invariant C.59

If *ack-from-syn* = `false` $\wedge\, mode_s \in \{$`rec`, `closed`$\}$ and there exists a SYN segment $(p, t) \in$ *in-transit*$_{cs}$ then $t \leq$ *first(open*$_s$*)*.

Proof: If the start state *ack-from-syn* is undefined, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

a = *receive-seg*$_{sc}$*(sn_s, ack_s, msg_s)* and a = *receive-seg*$_{sc}$*(sn_s, ack_s, msg_s, FIN)*

These steps may cause the premise of the invariant to go from false to true if $s.mode_c \in$ *sync-states*. Let (q, t') be the segment received in either step. From Invariant C.58 we know that if $mode_c \in$ *sync-states* then any SYN segment $(p, t) \in transit_{cs}$ has $t \leq t'$, and from Invariant C.22 we know that if $mode_s \in \{\texttt{rec}, \texttt{closed}\}$ then $t' \leq$ *first($open_s$)*. Therefore, the invariant holds after this step.

a = *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c)*, a = *shut-down*$_s$, and *time-out*$_s$

These steps may cause the premise of Part 1 to go from false to true. However, in all of these steps *first($open_s$)* is set to $now + \mu$. Since by Invariant C.1 we know $t \leq now + \mu$, we know the consequence is also true. Thus, Part 1 holds after these steps.

a = *crash*$_s$

This step may cause the premise of Part 1 to go from false to true. However, in this step *first($open_s$)* is set to $now + qt$. Since by Invariant C.1 we know $t \leq now + \mu$, we know the consequence is also true. ∎

Invariant C.60

If $mode_s = \texttt{listen} \ \wedge \ ack_c \in \texttt{BN}$ and there exists a SYN segment $(p, t) \in$ *in-transit*$_{cs}$ then *ack-from-syn* $= \texttt{true}$.

Proof: If the start state $mode_s = \texttt{closed}$, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

a = *passive-open*

This step may cause the premise of the invariant to go from true to false if $s.mode_s = \texttt{closed} \ \wedge now >$ *first($open_s$)*. If there is a SYN segment $(p, t) \in transit_{cs}$ then from Invariant C.1 we know $t \leq now$. Since by Invariant C.59 if *ack-from-syn* $= \texttt{false} \wedge mode_s = \texttt{closed}$ then *first($open_s$)* $\geq t$, we know that if this step causes the premise of the invariant to be true then *ack-from-syn* $= \texttt{true}$.

a = *receive-seg*$_{sc}$*(SYN, sn_s, ack_s)*

This step may cause the premise of the invariant to go from false to true if $s.mode_c = \texttt{syn-sent} \ \wedge [ack_s] = s.sn_c + 1$. However, this step also sets *ack-from-syn* to $\texttt{true}$.

$a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s)$ and $a = \textit{receive-seg}_{sc}(sn_s,\ ack_s,\ msg_s,\ FIN)$

These steps may cause the consequence of the invariant to go from true to false. However, from Invariant C.16 we know that if $mode_s = \texttt{listen}$ then $\textit{in-transit}_{sc}$ is empty, so if this step causes the consequence of the invariant to be false, the premise must also be false. ■

Invariant C.61

If $\textit{ack-from-syn} = \texttt{true}$ and there exists a SYN segment $(p, t) \in \textit{in-transit}_{cs}$ then $t \leq ltime_s(ack_c - 1) + \textit{clock-rate} + wt + rt + 2\mu$.

Proof: If the start state *ack-from-syn* is undefined, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a = \textit{receive-seg}_{sc}(SYN,\ sn_s,\ ack_s)$

This step may cause the premise of the invariant to go from false to true. Let (q, t') be the segment received in this step. This step causes ack_c to be assigned to $sn(q) + 1$. From Invariant C.38 we know that $t' \leq \textit{lst-time-cc}_s(sn(q)) + \textit{clock-rate} + wt + rt + \mu$, and from Invariant C.1 we know $t' \geq now$, and $t \leq now + \mu$. Therefore, we get $now \leq \textit{lst-time-cc}_s(sn(q)) + \textit{clock-rate} + wt + rt + \mu$, and $\textit{lst-time-cc}_s(sn(p)) \geq t - \textit{clock-rate} - wt - rt - 2\mu$, which means the invariant holds after this step. ■

Invariant C.62

If there exists a SYN segment $(p, t) \in \textit{in-transit}_{sc}$ then $\textit{lst-time-cc}_s(sn(p)) \leq t - \mu$.

Proof: If the start state $\textit{in-transit}_{cs} = \emptyset$, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$a = \textit{receive-seg}_{sc}(SYN,\ sn_s,\ ack_s)$

This step may cause the premise of the invariant to go from false to true. The invariant clearly holds after this step.

$a = recover_c$

This step may cause the consequence of the invariant to go from true to false. However, it is easy to see that there are no segments in $\textit{in-transit}_{cs}$ when this step is enabled.

$a = \textit{clock-counter-tick}_s$

This step may cause the consequence of the invariant to go from true to false by setting $\textit{lst-time-cc}_s(sn(p))$ to now if $s.\textit{clock-counter}_s = sn(p) - 1$. By Invariant C.30 we that $s.lst\text{-}time\text{-}cc(sn(p)-1) - s.lst\text{-}time\text{-}cc_s(sn(p)) \geq qt$, and from Invariant C.32 we know $s.lst\text{-}time\text{-}cc(sn(p)-1) \geq s.now - clock\text{-}rate$. Therefore, $s.now - clock\text{-}rate - qt \geq s.lst\text{-}time\text{-}cc_s(sn(p))$. Since we are assuming $s.lst\text{-}time\text{-}cc_s(sn(p)) \leq t - \mu$, we have $s.now - clock\text{-}rate - qt \geq t - \mu$ which means $t \leq now$. Thus, there cannot be a SYN segment in $s.in\text{-}transit_{sc}$. Since this step does not add any segments to $in\text{-}transit_{sc}$, we know the invariant holds after this step. ■

Invariant C.63

If $mode_s \in \{\texttt{rec}, \texttt{closed}\} \ \wedge \ ack\text{-}from\text{-}syn = \texttt{true}$ and there exists a SYN segment $(p,t) \in in\text{-}transit_{cs}$ then $lst\text{-}time\text{-}cc_s(ack_c - 1) \leq first(open_s) - \mu$.

Proof: If the start state $in\text{-}transit_{cs} = \emptyset$, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$\underline{a = receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c),\ a = shut\text{-}down_s, \text{ and } time\text{-}out_s}$

These steps may cause the premise of Part 1 to go from false to true. However, in all of these steps $first(open_s)$ is set to $now + \mu$ and since $lst\text{-}time\text{-}cc_s(x) \leq now$ for any $x \in \texttt{BN}$, the invariant holds after these steps.

$\underline{a = crash_s}$

This step may cause the premise of Part 1 to go from false to true. However, in this step $first(open_s)$ is set to $now + qt$, so we know the invariant is true after this step.

$\underline{a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)}$

This step may cause the premise of the invariant to go from false to true. Let (q, t') be the segment received in this step. This step causes ack_c to be assigned to $sn(q) + 1$. From Invariant C.62 we know that $lst\text{-}time\text{-}cc_s(sn(q)) \leq t' - \mu$, and from Invariant C.22 we know $t' \leq first(open_s)$. Therefore, the invariant holds after this step.

$\underline{a = recover_c}$

This step may cause the consequence of the invariant to go from true to false be setting $lst\text{-}time\text{-}cc_s(ack_c - 1)$ to now. It is easy to see that $s.lst\text{-}crash\text{-}time_s \geq s.lst\text{-}time\text{-}cc_s(ack_c - 1)$. Therefore, by Invariant C.55 we know that there are no SYN segments in $in\text{-}transit_{cs}$ is this step is enabled.

$\underline{a = clock\text{-}counter\text{-}tick_s}$

This step may cause the consequence of the invariant to go from true to false by setting

$lst\text{-}time\text{-}cc_s(ack_c - 1)$ to now if $s.clock\text{-}counter_s = ack_c - 2$. By Invariant C.55 we know that if $lst\text{-}crash\text{-}time_s \geq lst\text{-}time\text{-}cc_s(ack_c - 1)$ then there are no SYN segments in $in\text{-}transit_{cs}$, so the invariant holds for this case. By Invariant C.29 we know that if $lst\text{-}crash\text{-}time_s < lst\text{-}time\text{-}cc_s(ack_c - 1)$ then $s.lst\text{-}time\text{-}cc(ack_c - 2) - s.lst\text{-}time\text{-}cc_s(ack_c - 1) \geq ct - clock\text{-}rate$, and from Invariant C.32 we know $s.lst\text{-}time\text{-}cc(ack_c - 2) \geq s.now - clock\text{-}rate$. Therefore, $s.now - 2clock\text{-}rate - ct \geq s.lst\text{-}time\text{-}cc_s(ack_c - 1)$. From Invariant C.61 we know $t \leq ltime_s(ack_c - 1) + clock\text{-}rate + wt + rt + 2\mu$. Combining the two inequalities we get, $t \leq s.now$. Thus, there cannot be a SYN segment in $s.in\text{-}transit_{cs}$. Since this step does not add any segments to $in\text{-}transit_{cs}$, we know the invariant holds after this step. ■

Invariant C.64

If $mode_s = \texttt{listen}$ and there exists a SYN segment $(p, t) \in in\text{-}transit_{cs} \wedge ack_c \in \texttt{BN}$ then $clock\text{-}counter_s > ack_c - 1$.

Proof: If the start state $mode_s = \texttt{closed}$, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below. The step with $a = receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$ is not critical even thought it may cause the assigning of ack_c to $[sn_s] + 1$. This assignment is made if $mode_c = \texttt{syn-sent} \wedge [ack_s] = s.sn_c + 1$. However, from Invariant C.52 we know that if this step causes this assignment, $mode_s \neq \texttt{listen}$, so it cannot cause the premise of the invariant to be true.

<u>$a = passive\text{-}open$</u>

This step cause the premise of the invariant to go from false to true, if $s.mode_s = \texttt{closed}$, $s.now > first(open_s)$, and there is a SYN segment $(p, t) \in in\text{-}transit_{cs} \wedge s.ack_c \in \texttt{BN}$. If $s.clock\text{-}counter_s = ack_c - 1$ then by Invariant C.32 $s.lst\text{-}time\text{-}cc_s(s.clock\text{-}counter_s) \geq s.now - clock\text{-}rate$. However, this contradicts Invariant C.63 which says $lst\text{-}time\text{-}cc_s(ack_c - 1) \leq first(open_s) - \mu$. Therefore, we know that after this step $s'.clock\text{-}counter_s = s'.ack_c - 1$.

If $s.clock\text{-}counter_s < ack_c - 1$ and $lst\text{-}crash\text{-}time_s < ltime_s(ack_c - 1)$ then by Invariant C.29 $lst\text{-}time\text{-}cc_c(s'.clock\text{-}counter_s) - lst\text{-}time\text{-}cc_s(s'.ack_c - 1) \geq ct/2$. We also know from Invariant C.32 that $lst\text{-}time\text{-}cc_s(clock\text{-}counter_s) \geq now - clock\text{-}rate$. Therefore, $lst\text{-}time\text{-}cc_c(ack_c - 1) \leq now - clock\text{-}rate - ct/2$. From Invariants C.60 and C.61 we know that $t \leq ltime_s(ack_c - 1) + clock\text{-}rate + wt + rt + 2\mu$. Combining the two inequalities we get $t \leq now - ct/2 + wt + rt + 2\mu$. Since, $ct/2 > wt + rt + 2\mu$ we get $t < now$, which contradicts Invariant C.1. Therefore, if $lst\text{-}crash\text{-}time_s < ltime_s(ack_c - 1)$ the invariant holds after this step. If $lst\text{-}crash\text{-}time_s \geq ltime_s(ack_c - 1)$ then by Invariant C.55 we know there are no SYN segments in $in\text{-}transit_{cs}$, so the premise does not become true in this situation.

$\underline{a = clock\text{-}counter\text{-}tick_s}$

This step may cause the consequence of the invariant to go from true to false if incrementing $clock\text{-}counter_s$ causes $s'.clock\text{-}counter_s < ack_c$. The proof that the premise must also be false in this situation is the same as the proof that the invariant holds after step $a =$ *passive-open*. ■

Invariant C.65

If $mode_s = \texttt{listen}$ and there exists a SYN segment $(p, t) \in in\text{-}transit_{cs}$ then for any non-SYN segment $(q, t') \in in\text{-}transit_{cs}$ $clock\text{-}counter_s > ack(q) - 1$.

Proof: If the start state $mode_s = \texttt{closed}$, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

$\underline{a = send\text{-}seg_{cs}(SYN,\ sn_c)}$

This step can make the premise of Part 1 to go from false to true by adding a SYN segment to $in\text{-}transit_{cs}$. However, from Invariant C.24 we know that if $mode_c = \texttt{syn-sent}$ then there are only SYN segments in $in\text{-}transit_{cs}$, so the invariant holds after this step.

$\underline{a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c) \text{ and } a = send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c,\ FIN)}$

These steps may cause the consequence of Part 1 to go from true to false by adding a non-SYN segment to $in\text{-}transit_{cs}$. By Invariants C.60 and C.64 we know that if these steps cause the consequence of the invariant to be false, then the premise must also be false.

$\underline{a = passive\text{-}open}$

The proof that the invariant holds after this step is very similar to the proof that Invariant C.64 holds after the same step.

$\underline{a = clock\text{-}counter\text{-}tick_s}$

The proof for this set is also very similar to the proof for the same step for Invariant C.64.

■

■

Invariant 8.7

If $mode_s = \texttt{syn-rcvd} \wedge new\text{-}isn_s = \texttt{true} \wedge\ ack_c \in \texttt{BN}$ then $sn_s \geq ack_c$.

Proof: This invariant follows from Invariant C.64. ■

Invariant 8.8

If $mode_s = \texttt{syn-rcvd} \wedge new\text{-}isn_s = \texttt{true}$ then for all segments $(p,t) \in in\text{-}transit_{cs}$, $ack(p) < sn_s + 1$.

Proof: This invariant follows from Invariant C.65. ■

Invariant 8.9

If $mode_c = \texttt{syn-sent}$ then for all SYN segments $(p,t) \in in\text{-}transit_{sc}$ such that $ack(p) = sn_c + 1$, $sn(p) \geq ack(q)$ for all $(q,t') \in in\text{-}transit_{cs}$.

Proof: . From Invariant C.24 we know that if $mode_c = \texttt{syn-sent}$ then there can only be SYN segments in $in\text{-}transit_{cs}$. Since, SYN segments sent by the client do not have acknowledgment numbers, the invariant holds. ■

Invariant C.66

If $mode_s \in sync\text{-}states$ and there exists a SYN segment $(p,t) \in in\text{-}transit_{sc}$ with $ack(p) = sn_c + 1$ then $mode_c \neq \texttt{syn-sent}$.

Proof: If the start state $mode_s = \texttt{closed}$, so the invariant holds in this state. We consider critical steps of the form (s, a, s') below.

a = *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c)* and a = *receive-seg*$_{cs}$*(sn_c, ack_c, msg_c, FIN)*

These steps may cause the premise of the invariant to go from false to true. However, from Invariant C.24 we know that if there are non-SYN segments in $in\text{-}transit_{cs}$ then $mode_c \neq \texttt{syn-sent}$, so the invariant holds after these steps.

a = *prepare-msg*$_c$

This step may cause the premise of the invariant to go from false to true by incrementing sn_c. However, this step is enabled only if $mode_c \neq \texttt{syn-sent}$, so the invariant holds after this step.

a = *send-msg*$_c$*(open, m, close)*

This step can make the consequence of the invariant false. However, by Invariant 8.6 we know the premise is also false after this step. ■

Invariant 8.10

If $mode_c$ = `syn-sent` and there exists SYN segment $(p,t) \in$ *in-transit*$_{sc}$ such that $ack(p) = sn_c + 1$ then $sn(p) \geq sn(q)$ for all non-SYN segments $(q,t') \in$ *in-transit*$_{sc}$.

Proof: In the start state both $mode_c$ = `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

a = *send-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*

This step may cause the premise of the invariant to go from false to true. However, this step is enabled if $mode_s$ = `syn-rcvd`. From Invariant C.24 we know there are only SYN segments in *in-transit*$_{sc}$ if $mode_s$ = `syn-rcvd`, so the invariant holds after this step.

a = *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*)* and a = *send-seg*$_{sc}$*(*sn_s*,* ack_s*,* msg_s*, FIN)*

These steps may cause the consequence of the invariant to go from true to false by adding a non-SYN segment to *in-transit*$_{sc}$. These steps are enabled if $mode_s \in$ *sync-states*. From Invariant C.66 we know if these steps cause the consequence to be false, then the premise is also false.

a = *prepare-msg*$_c$

This step may cause the premise of the invariant to go from false to true by incrementing sn_c. However, this step is enabled only if $mode_c \neq$ `syn-sent`, so the invariant holds after this step.

Invariant C.67

1. If $sn_c = sn(p) + 1 \vee sn_c = sn(p) + 2$ for any segment $(p,t) \in$ *in-transit*$_{cs}$ then *lst-time-sn*$_c(sn_c) \geq t - \mu$.

2. If $sn_s = sn(p)+1 \vee sn(p)+2$ for any segment $(p,t) \in$ *in-transit*$_{sc}$ then *lst-time-sn*$_s(sn_s) \geq t - \mu$.

Proof: If the start state both *in-transit*$_{cs}$ and *in-transit*$_{sc}$ are empty, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

1. a = *prepare-msg*$_c$

   This step may cause the premise to the invariant to go from false to true. However in this step *lst-time-sn*$_c(sn_c)$ is set to *now*, so by Invariant C.1 we know Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ∎

Invariant C.68

1. If $sn_c = sn(p) + i$ for any segment $(p, t) \in$ *in-transit*$_{cs}$ and $2 < i < 2^{32}$ then *lst-time-sn*$_c(sn_c) \geq t - \mu + (i - 2) \times$ *data-rate*.

2. If $sn_s = sn(p) + i$ for any segment $(p, t) \in$ *in-transit*$_{sc}$ and $2 < i < 2^{32}$ then *lst-time-sn*$_s(sn_s) \geq t - \mu + (i - 2) \times$ *data-rate*.

Proof: For this invariant we have two levels of induction. The first level is induction on i, and the second level is the induction on the steps of $\mathcal{BTCP}^h$. The base case of the induction on i is for $i = 2$. This case is Invariant C.67. For the inductive case we assume that the Invariant holds for j and show it holds for $j + 1$. We show it holds for $j + 1$ by induction on the steps of $\mathcal{BTCP}^h$. We consider critical actions of the form (s, a, s') below.

1. $a =$ *prepare-msg*$_c$

   This step may cause the premise of Part 1 to go from false to true is $s.sn_c = sn(p) + j$. This step is enabled if $s.now \geq$ *s.first(prep-msg*$_c$*)*. From Invariant C.27 we know that *s.first(prep-msg*$_c$*)* $\geq$ *s.lst-time-sn*$(s.sn_c)$ + *data-rate*. Therefore, since we assume the invariant holds for j, then it clearly holds for $j + 1$.

2. The proof for Part 2 is symmetric to the proof for Part 1. ∎

Invariant C.69

1. If $mode_c \neq$ `closed` and there exists a segment $(p, t) \in$ *in-transit*$_{cs}$ then $sn_c \geq sn(p)$.

2. If $mode_s \neq$ `closed` and there exists a segment $(p, t) \in$ *in-transit*$_{sc}$ then $sn_s \geq sn(p)$.

Proof: In the start state both $mode_c =$ `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below. The step with $a =$ *send-msg*$_c$*(open, m, close)* is not critical because if $mode_c =$ `closed` $\wedge$ *now* $>$ *first(open*$_c$*)* then there are no segments in *in-transit*$_{cs}$.

1. $a =$ *send-seg*$_{cs}$*(p)*

   These steps may casue the cause the premise of the invariant to go from false to true, but $sn_c = sn(p)$, so the invariant holds after these steps.

a = *prepare-msg*$_c$

This step may cause the consequence of the invariant to go from true to false by incrementing sn_c. Since this step also sets $lst\text{-}time\text{-}sn(s.sn_c)$ to now, if the consequence becomes false then by Invariant C.68 $now \geq t - \mu + (2^{31} \times data\text{-}rate)$. Since $(2^{31} \times data\text{-}rate) > \mu$, we get $now \geq t$. However, this violates Invariant C.1. Therefore there can be no such segment in $in\text{-}transit_{cs}$, so Part 1 holds after this step.

2. The proof for Part 2 is symmetric to the proof for Part 1. ■

Invariant 8.11

If $mode_s =$ `syn-rcvd` $now \leq wait\text{-}t\_o_s \ \wedge mode_c \neq$ `closed` then $ack_s \leq sn_c + 1$.

Proof: In the start state both $mode_c =$ `closed`, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

a = *send-msg*$_c$*(open, m, close)*

This step may cause the premise of the invariant to go from false to true. From Invariant 8.5 we know that the consequence is also true after this step.

a = *receive-seg*$_{cs}$*(SYN,* sn_c*)*

This step may cause the premise of the invariant to go from false to true. However, from Invariant C.69 we know that $sn_c \geq sn(p)$ for any SYN segment $(p, t) \in in\text{-}transit_{cs}$. Therefore, the invariant holds after this step.

a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*

These steps may cause the consequence of the invariant to go from true to false by incrementing ack_s. However, ack_s is only incremented if $mode_s \in sync\text{-}states$, so the invariant holds after this step.

a = *prepare-msg*$_c$

This step may cause the consequence of the invariant to go from true to false by incrementing sn_c.

Invariant C.70

If $mode_c =$ `syn-sent` and there exists a SYN segment $(p, t) \in in\text{-}transit_{sc}$ such that $ack(p) = sn_c + 1$ then $mode_s \notin sync\text{-}states$.

Proof: In the start state $mode_c = \texttt{closed}$, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

a = *send-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*

This step may cause the premise of the invariant to go from true to false. However, this step is only enabled if $mode_s = \texttt{syn-rcvd}$, so the invariant holds after this step.

a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*

These steps may cause the consequence of the invariant to go from true to false. However, from Invariant C.24 we know that if there are non-SYN segments in *in-transit*$_{cs}$ then $mode_c \neq \texttt{syn-sent}$, so the invariant holds after these steps. ∎

Invariant C.71

If $mode_c = \texttt{syn-sent} \ \wedge mode_s \notin \{\texttt{closed}, \texttt{listen}\}$ and there exists a SYN segment $(p, t) \in$ *in-transit*$_{sc}$ such that $ack(p) = sn_c + 1$ then $sn_s = sn(p) \ \wedge ack_s = ack(p)$

Proof: In the start state $mode_c = \texttt{closed}$, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

a = *send-seg*$_{sc}$*(SYN,* sn_s*,* ack_s*)*

This step may cause the premise of the invariant to go from true to false. However, it is clear that after this step $sn_s = sn(p) \ \wedge ack_s = ack(p)$.

a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*)* and a = *receive-seg*$_{cs}$*(*sn_c*,* ack_c*,* msg_c*, FIN)*

These steps may cause the consequence of the invariant to go from true to false by incrementing ack_s. However, from Invariant C.24 we know that if there are non-SYN segments in *in-transit*$_{cs}$ then $mode_c \neq \texttt{syn-sent}$, so the invariant holds after these steps.

a = *prepare-msg*$_s$

This step may cause the consequence of the invariant to go from true to false. However, this step is only enable if $mode_s \in$ *sync-states*, and from Invariant C.70 we know that if the premise of the invariant is true then $mode_s \notin$ *sync-states*. Therefore, if $mode_s \in$ *sync-states* then the premise of the invariant must be false. ∎

Invariant 8.12

If *just-estb* $= \texttt{true} \ \wedge sn_s \in \texttt{BN}$ then $ack_c > sn_s$.

In the start state *just-estb* is undefined, so the invariant holds in this state. We consider critical actions of the form (s, a, s') below.

a = *receive-seg*$_{sc}$*(SYN, sn_s, ack_s)*

This step may cause the premise of the invariant to go from true to false. This steps also assigns ack_c to $[sn_s]+1$. From Invariant C.71 we know $[sn_s] = sn_s$, so the invariant holds after this step.

a = *receive-seg*$_{sc}$*(sn_s, ack_s, msg_s)* and a = *receive-seg*$_{sc}$*(sn_s, ack_s, msg_s, FIN)*

These steps may cause the consequence of the invariant to go from true to false, but after these steps *just-estb* = `false`, so the invariant holds.

a = *prepare-msg*$_s$

This step may cause the consequence of the invariant to go from true to false. However, *just-estb* = `false` after this step, so the invariant holds. ■

Invariant 8.13

1. If $mode_c \in$ *sync-states* then for all segments $(p, t) \in$ *in-transit*$_{cs}$, $sn_c \geq sn(p)$.

2. If $mode_s \in$ *sync-states* then for all segments $(p, t) \in$ *in-transit*$_{sc}$, $sn_s \geq sn(p)$.

Proof: This invariant follows from Invariant C.69. ■

Invariant 8.14

1. If $mode_c \in$ *sync-states* $\wedge\ ack_s \in$ `BN` then $sn_c + 1 \geq ack_s$.

2. If $mode_s \in$ *sync-states* $\wedge\ ack_c \in$ `BN` then $sn_s + 1 \geq ack_c$.

Proof: The proof of this Invariant is similar to the proof of Invariant 7.2. ■

Invariant 8.15

1. If $mode_c \in$ *sync-states* $\wedge$ *new-sn*$_c$ = `true` then for all segments $(p, t) \in$ *in-transit*$_{sc}$, $sn_c + 1 > ack(p)$.

2. If $mode_s \in$ *sync-states* $\wedge$ *new-sn*$_s$ = `true` then for all segments $(p, t) \in$ *in-transit*$_{cs}$, $sn_s + 1 > ack(p)$.

Proof: The proof of this Invariant is similar to the proof of Invariant 7.3. ■

Invariant 8.16

1. If $mode_c \in sync\text{-}states$ then for all $(p,t) \in in\text{-}transit_{cs}$, $ack_c \geq ack(p)$.

2. If $mode_s \in sync\text{-}states$ then for all $(p,t) \in in\text{-}transit_{sc}$, $ack_s \geq ack(p)$.

Proof: The proof of this Invariant is similar to the proof of Invariant 7.29. ■

Invariant 8.17

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states \wedge mode_c \in \{\texttt{rec}, \texttt{reset}\} \cup sync\text{-}states$ and there exists $(p,t) \in in\text{-}transit_{cs}$ such that $sn(p) \geq ack_s$, then $sn_c = sn(p)$.

2. If $mode_c \in sync\text{-}states$ and there exists $(p,t) \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, then $sn_s = sn(p)$.

Proof: The proof for this invariant is similar to the proof for Invariant 7.53.

Invariant 8.18

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states$ and there exists $(p,t) \in in\text{-}transit_{cs}$ such that $sn(p) \geq ack_s$, then for all other non-SYN segments $(q,t') \in in\text{-}transit_{cs}$, $sn(q) \leq sn(p)$.

2. If $mode_c \in sync\text{-}states$ and there exists $(p,t) \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, then for all other non-SYN segments $(q,t') \in in\text{-}transit_{sc}$, $sn(q) \leq sn(p)$.

Proof: The proof for this invariant is similar to the proof for Invariant 7.62.

Appendix D

Invariance proofs for $\mathcal{TTCP}^h$

As we did in Appendix B, we use the standard inductive technique for proving the invariants.

Invariant 10.7

1. For all segments $p \in \textit{in-transit}_{cs}$, $sn_c \geq sn(p)$.

2. For all segments $p \in \textit{in-transit}_{sc}$, $sn_s \geq sn(p)$.

3. For all segments $p \in \textit{in-transit}_{cs}$, $cc\_send \geq cc\_send(p)$. ■

Proof: The same as the proof of Invariant 10.7.

Invariant 10.8

1. If $mode_c \neq \texttt{closed}$ then $cc\_send = id_c$.

2. If $mode_s \in \textit{sync-states}$ then $cc\_rcvd = id_s$.

3. If there exists a segment $p \in \textit{in-transit}_{cs}$ such that $cc\_send(p) = k$, then $k \in \textit{used-id}_c \cup \{\textit{crash-id}_c\}$. ■

Proof: Straighforward.

Invariant 10.9

1. For all $i \in \mathtt{N} \cup \{\texttt{nil}\}$, $(i, \texttt{nil}) \notin \textit{estb-cc}$.

2. For all $j \in \mathtt{N} \cup \{\texttt{nil}\}$, $(\texttt{nil}, j) \notin \textit{estb-cc}$.

3. For all $i \in \mathtt{N} \cup \{\mathtt{nil}\}$, $(i, \mathtt{nil}) \notin assoc$.

4. For all $j \in \mathtt{N} \cup \{\mathtt{nil}\}$, $(\mathtt{nil}, j) \notin assoc$. ∎

Proof: Straightforward.

Invariant 10.10

1. If $mode_s \in \{\mathtt{listen}, \mathtt{syn\text{-}rcvd}\}$ then $rcv\text{-}buf_s = \epsilon$.

2. If $mode_s \in \{\mathtt{listen}, \mathtt{syn\text{-}rcvd}\}$ then $msg_s = \mathtt{null}$.

3. If $mode_s \in \{\mathtt{listen}, \mathtt{syn\text{-}rcvd}\}$ then $last\text{-}msg_s = \mathtt{null}$.

4. If $mode_s \in sync\text{-}states$ then $temp\text{-}data = \mathtt{null}$.

Proof: Straightforward. ∎

Invariant 10.11

1. If $mode_s = \mathtt{listen}$ and there exists a segment $p \in in\text{-}transit_{cs}$ such that $cc\_send(p) > cache\_cc$ then $(cc\_send(p), cc\_send(p)) \notin assoc$.

2. If $mode_c \in \{\mathtt{syn\text{-}sent}, \mathtt{syn\text{-}sent*}\}$ then for all j, $(id_c, j) \notin estb\text{-}cc$.

3. If $mode_c \in \{\mathtt{syn\text{-}sent}, \mathtt{syn\text{-}sent*}\}$ and there exists a segment p of the form *(SYN, cc_rcvd, sn_s, ack_s)* or *(SYN, cc_rcvd, sn_s, ack_s, FIN)* in $in\text{-}transit_{sc}$ such that $cc\_rcvd(p) = cc\_send$ and $ack(p) = sn_s + 1$ then for all j, $(id_c, j) \notin assoc$.

4. If $mode_s = \mathtt{syn\text{-}rcvd} \ \wedge choose\text{-}isn_s$ then for all k, $(k, isn_s) \notin estb\text{-}cc$.

Invariant 10.12

1. If $mode_s = \mathtt{syn\text{-}rcvd}$ and there exists a segment $p \in in\text{-}transit_{cs}$ such that $cc\_send(p) = cc\_rcvd \ \wedge ack(p) = sn_s + 1$, then $(cc\_send(p), isn_s) \in estb\text{-}cc$.

2. If $mode_c \in sync\text{-}states$ or there exists a segment p of type *(SYN, cc_rcvd, sn_s, ack_s, msg_s,)* or *(SYN, cc_rcvd, sn_s, ack_s, msg_s, FIN)* with $cc\_rcvd(p) = ccsend$ then $(id_c, id_c) \in assoc$.

3. If $mode_s \in$ *sync-states* then $(id_s, id_s) \in assoc$.

4. If $(k, id_s) \in assoc$ and there exists j such that $(k, j) \in$ *estb-cc* then $j = isn_s$.

Proof:

Invariant 10.13

If $mode_s \in$ *sync-states* $\wedge$ *cc_send* $=$ *cc_rcvd* and there exists a non-SYN segment $p \in$ *in-transit*$_{cs}$ with *cc_send*$(p) =$ *cc_rcvd* $\wedge$ $ack(p) = sn_s + 1$ then $mode_c \notin \{$`syn-sent`, `syn-sent*`$\}$. ∎

Invariant 10.14

If $mode_c \in \{$`syn-sent`, `syn-sent*`$\}$ and there exists a SYN segment $p \in$ *in-transit*$_{sc}$ such that *cc_rcvd*$(p) =$ *cc_send* $\wedge ack(p) = sn_c{+}1 \wedge sn(p) = isn_s$ then $mode_s \in \{$`syn-rcvd`, `rec`, `reset`$\}$.
∎

Invariant 10.15

1. If $mode_c \in \{$`syn-sent*`, `fin-wait-1`, `fin-wait-2`, `closing`, `timed-wait`, `last-ack`$\}$ then *send-buf*$_c = \epsilon \wedge$ *rcvd-close*$_c =$ `true`.

2. If $mode_s \in \{$`fin-wait-1`, `fin-wait1*`, `fin-wait-2`, `closing`, `closing*`, `timed-wait`, `last-ack`, `last-ack*`$\}$ then *send-buf*$_s = \epsilon \wedge$ *rcvd-close*$_s =$ `true`. ∎

Proof: The proof is the same as for Invariant 7.13. ∎

Invariant 10.16

If *temp-data* $\neq$ `null` $\wedge\, msg_c \neq$ `null` $\wedge ((id_c, isn_s) \in$ *estb-cc* $\vee (id_c, id_s) \in assoc) \wedge sn_c < ack_s$ then $msg_c =$ *temp-data*. ∎

Invariant 10.17

If $mode_c \in \{$`syn-sent`, `syn-sent*`$\}$ $\wedge$ *temp-data* $\neq$ `null` and there exists a SYN segment

$p \in \mathit{in\text{-}transit}_{sc}$ such that $\mathit{cc\_rcvd}(p) = \mathit{cc\_send} \ \wedge \mathit{ack}(p) = \mathit{sn}_c + 1 \ \wedge \ \mathit{sn}(p) = \mathit{isn}_s$ then $\mathit{msg}_c = \mathit{temp\text{-}data}$. ■

Invariant 10.18

1. If $\mathit{rcv\text{-}buf}_c = \epsilon \ \wedge \ \mathit{last\text{-}msg}_s \neq \texttt{null} \ \wedge \ \mathit{msg}_s \neq \texttt{null} \ \wedge \ (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_s < \mathit{ack}_c$ then $\mathit{msg}_s = \mathit{last\text{-}msg}_c$.

2. If $\mathit{rcv\text{-}buf}_s = \epsilon \ \wedge \ \mathit{last\text{-}msg}_c \neq \texttt{null} \ \wedge \ \mathit{msg}_c \neq \texttt{null} \ \wedge \ (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_c < \mathit{ack}_s$ then $\mathit{msg}_c = \mathit{last\text{-}msg}_s$.

Invariant 10.19

1. If $\mathit{rcv\text{-}buf}_c \neq \epsilon \ \wedge \ \mathit{msg}_s \neq \texttt{null} \ \wedge \ (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_s < \mathit{ack}_c$ then $\mathit{msg}_s = \mathit{last}(\mathit{rbuf}_c)$.

2. If $\mathit{rcv\text{-}buf}_s \neq \epsilon \ \wedge \ \mathit{msg}_c \neq \texttt{null} \ \wedge \ (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_c < \mathit{ack}_s$ then $\mathit{msg}_s = \mathit{last}(\mathit{rbuf}_s)$.

Proof:

Invariant 10.20

1. If $\mathit{last\text{-}msg}_c = \texttt{null} \wedge \mathit{msg}_s \neq \texttt{null} \wedge (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_s < \mathit{ack}_c$ then $\mathit{rcv\text{-}buf}_c \neq \epsilon$.

2. If $\mathit{last\text{-}msg}_s = \texttt{null} \wedge \mathit{msg}_c \neq \texttt{null} \wedge (\mathit{id}_c, \mathit{id}_s) \in \mathit{assoc} \wedge \mathit{sn}_c < \mathit{ack}_s$ then $\mathit{rcv\text{-}buf}_s \neq \epsilon$.

Invariant 10.21

1. If $\mathit{msg}_c \neq \texttt{null}$ and there exists $p \in \mathit{in\text{-}transit}_{cs}$ such that $\mathit{sn}(p) = \mathit{sn}_c$ then $\mathit{msg}(p) = \mathit{msg}_c$.

2. If $\mathit{msg}_s \neq \texttt{null}$ and there exists $p \in \mathit{in\text{-}transit}_{sc}$ such that $\mathit{sn}(p) = \mathit{sn}_s$ then $\mathit{msg}(p) = \mathit{msg}_s$. ■

Proof: Same as the proof of Invariant 7.36.

Invariant 10.22

1. If there exists segments p and q on $in\text{-}transit_{cs}$ such that $sn(p) = sn(q) \ \wedge msg(p) \neq$ `null` $\wedge \ msg(q) \neq$ `null` then $msg(p) = msg(q)$.

2. If there exists segments p and q on $in\text{-}transit_{sc}$ such that $sn(p) = sn(q) \ \wedge msg(p) \neq$ `null` $\wedge \ msg(q) \neq$ `null` then $msg(p) = msg(q)$. ■

Proof: Same as the proof of Invariant 7.37.

Invariant 10.23

1. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states$ and there exists $p \in in\text{-}transit_{cs}$ such that $sn(p) = ack_s$ then $msg(p) \neq$ `null`.

2. If $mode_c \in sync\text{-}states$ and there exists $p \in in\text{-}transit_{sc}$ such that $sn(p) = ack_c$ then $msg(p) \neq$ `null`. ■

Proof: Same as the proof of Invariant 7.55.

Invariant 10.24

If $ack_c \in \mathbb{N}$ then for all $p \in in\text{-}transit_{cs}$, $ack_c \geq ack(p)$. ■

Proof: Same as the proof for Invariant 7.23.

Invariant 10.25

If $mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\} \ \wedge \ (id_c, id_s) \in assoc \ \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\}$ then for all segments $p \in in\text{-}transit_{sc}$, $ack_s \geq ack(p)$. ■

Invariant 10.26

1. If there exists a SYN segment $p \in in\text{-}transit_{cs}$ such that $cc\_send(p) = cc\_send$ and $cc\_send(p) > cache\_cc$ then $sn_c = sn(p)$.

2. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states \wedge mode_c \in \{\texttt{rec}, \texttt{reset}\} \cup sync\text{-}states$ and there exists $p \in in\text{-}transit_{cs}$ such that $(cc\_send(p), isn_s) \in estb\text{-}cc \ \wedge \ cc\_send(p) = cc\_rcvd \ \wedge$ $sn(p) \geq ack_s$, then $sn_c = sn(p)$.

3. If $mode_c \in sync\text{-}states \wedge (isn_c, isn_s) \in assoc$ and there exists $p \in in\text{-}transit_{sc}$ such that $sn(p) \geq ack_c$, then $sn_s = sn(p)$.

Proof: In the start state $in\text{-}transit_{cs}$ is empty. We consider critical steps of the form (s, a, s') below.

1. $\underline{a = send\text{-}msg_c(open,\ m,\ close)}$

   This step may cause the consequence of the invariant to go from true to false by incrementing sn_c. However, since cc_send is also incremented in this step, it is clear that there are no segments in $p \in in\text{-}transit_{cs}$ with $cc\_send(p) = cc\_send$, so the premise of the invariant is also false.

   $\underline{a = send\text{-}seg_{cs}(SYN,\ cc\_send,\ sn_c,\ msg_c)}$ and
   $\underline{a = send\text{-}seg_{cs}(SYN,\ cc\_send,\ sn_c,\ msg_c,\ FIN)}$

   These steps may cause the premise of the invariant to go from false to true by adding a SYN segment to $in\text{-}transit_{cs}$. However, on these segments $sn_c = sn(p)$, so the invariant holds after these steps.

   $\underline{a = prepare\text{-}msg_c}$.
   This step may also cause the consequence of the invariant to go from true to false by incrementing sn_c. However, if this step is enabled, we know from Invariant **??** than $s.cc\_send \leq s.cache\_cc$ and since $s.cc\_send \geq cc\_send(p)$ for any $p \in transit_{cs}$, the premise of the invariant must also be false.

2.

3. ∎

Invariant 10.27

1. If $mode_c \in \{$`close-wait`, `closing`, `last-ack`, `timed-wait`$\} \wedge mode_s \notin \{$`rec`, `reset`$\}$ $\wedge\ (id_c, id_s) \in assoc$ then $mode_s \in \{$`fin-wait-1`, `fin-wait1*`, `fin-wait-2`, `closing`, `closing*`, `timed-wait`, `last-ack`, `last-ack*`$\}$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{close-wait*}, \texttt{closing}, \texttt{closing*}, \texttt{last-ack}, \texttt{last-ack*}, \texttt{timed-wait}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge \ (id_c, isn_s) \in estb\text{-}cc \ \vee \ (id_c, id_s) \in assoc$ then $mode_c \in \{\texttt{syn-sent*}, \texttt{fin-wait-1}, \texttt{fin-wait-2}, \texttt{closing}, \texttt{timed-wait}, \texttt{last-ack}\}$.

■

Invariant 10.28

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, id_s) \in assoc$ then $sn_s < ack_c$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{close-wait*}, \texttt{closing}, \texttt{closing*}, \texttt{last-ack}, \texttt{last-ack*}, \texttt{timed-wait}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, isn_s) \in estb\text{-}cc \ \vee \ (id_c, id_s) \in assoc$ then $sn_c < ack_s$.

Invariant 10.29

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists l such that $(id_c, l) \in assoc$ then for all non-SYN segments $p \in in\text{-}transit_{sc}$, $sn(p) < ack_c$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{close-wait*}, \texttt{closing}, \texttt{closing*}, \texttt{last-ack}, \texttt{last-ack*}, \texttt{timed-wait}\}$ and there exists k, such that $(k, isn_s) \in estb\text{-}cc \ \vee (k, id_s) \in assoc$ then for all non-SYN segments $p \in in\text{-}transit_{cs}$, $sn(p) < ack_c$.

Invariant 10.30

1. If $mode_c \in \{\texttt{syn-sent}, \texttt{syn-sent*}\}$ and there exists j such that $(id_c, j) \in assoc$ and there exists a SYN segment $p \in in\text{-}transit_{sc}$ such that $cc\_rcvd(p) = cc\_send$ then for all segments $q \in in\text{-}transit_{sc}$ such that $cc\_rcvd(q) = cc\_send$, $sn(q) \leq sn(p)$.

2. If $mode_c \in sync\text{-}states$ and there exists j such that $(id_c, j) \in assoc$ and there exists a non-SYN segment $p \in in\text{-}transit_{sc}$ such that $cc\_rcvd(p) = cc\_send \ \wedge \ sn(p) \geq ack_c$, then for all non-SYN segments $q \in in\text{-}transit_{sc}$ $sn(q) \leq sn(p)$.

3. If $mode_s \in \{\texttt{syn-rcvd}\} \cup sync\text{-}states$ and there exists i, such that $i = isn_c^s \wedge (i, isn_s) \in estb\text{-}pairs$ and there exists a non-SYN segment $p \in in\text{-}transit_{cs}$ such that $sn(p) \geq ack_s$, then for all non-SYN segments $q \in in\text{-}transit_{cs}$ $sn(q) \leq sn(p)$.

Invariant 10.31

1. If $mode_c = \texttt{last-ack} \wedge mode_s \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, id_s) \in assoc$ then $rcv\text{-}buf_c = \epsilon$.

2. If $mode_s \in \{\texttt{last-ack}, \texttt{last-ack*}\} \wedge mode_c \notin \{\texttt{rec}, \texttt{reset}\} \wedge (id_c, isn_s) \in estb\text{-}cc \vee (id_c, id_s) \in assoc$ then $rcv\text{-}buf_s = \epsilon$.

Invariant 10.32

1. If $mode_c = \texttt{last-ack}$ and there exists l such that $(id_c, l) \in assoc$ then $rcv\text{-}buf_c = \epsilon$.

2. If $mode_s \in \{\texttt{last-ack}, \texttt{last-ack*}\}$ and there exists k, such that $(k, isn_s) \in estb\text{-}cc \vee (k, id_s) \in assoc$ then $rcv\text{-}buf_s = \epsilon$.

Invariant 10.33

1. If $mode_c \in \{\texttt{close-wait}, \texttt{closing}, \texttt{last-ack}, \texttt{timed-wait}\}$ and there exists l such that $(id_c, l) \in assoc$ then $rcvd\text{-}close_s = \texttt{true} \vee id_s \neq l$.

2. If $mode_s \in \{\texttt{close-wait}, \texttt{close-wait*}, \texttt{closing}, \texttt{closing*}, \texttt{last-ack}, \texttt{last-ack*}, \texttt{timed-wait}\}$ and there exists k such that $(k, isn_s) \in estb\text{-}cc \vee (k, id_c) \in assoc$ then $rcvd\text{-}close_c = \texttt{true} \vee id_c \neq k$.

Bibliography

[1] Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Yehuda Afek, Hagit Attiya, Alan Fekete, Michael Fischer, Nancy Lynch, Yishay Mansour, Da-Wei Wang, and Lenore Zuck. Reliable communication over unreliable channels. *Journal of the ACM*, 41(6):1267–1297, November 1994.

[3] Hagit Attiya, Slomo Dolev, and Jennifer Welch. Connection management without retaining information. Technical Report LPCR 9316, Laboratory for Parallel Computing Research, Dept. of Computer Science, The Technion, June 1993.

[4] D. Belsnes. Single message communication. *IEEE Transactions on Communications*, 24(2), February 1976.

[5] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), 1984.

[6] Robert Braden. Extending TCP for transactions – concepts. Internet RFC-1379, November 1992.

[7] Robert Braden. T/TCP – TCP extensions for transactions – functional specification. Internet RFC-1644, July 1994.

[8] Robert Braden and David Clark. Transport protocols for transactions and streaming. Unpublished manuscript, March 1993.

[9] Douglas Comer. *Internetworking with TCP/IP*, volume 1. Prentice Hall, second edition, 1991.

[10] Alan Fekete, Nancy Lynch, Yishay Mansour, and John Spinelli. The impossibility of implementing reliable communication in the face of crashes. *Journal of the ACM*, 40(5):1087–1107, November 1993.

[11] S.J. Garland and J.V. Guttag. A guide to the larch prover. Technical Report 82, DEC, Systems Research Center, December 1991.

[12] Rainer Gawlick, Roberto Segala, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139, December 1993.

[13] Rainer Gawlick, Roberto Segala, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. In Serge Abiteboul and Eli Shamir, editors, *Automata, Languages and Programming* (21st International Colloquium, ICALP'94, Jerusalem, Israel, July 1994), volume 820 of *Lecture Notes in Computer Science*, pages 166–177. Springer-Verlag, 1994. Full version in [12]. Also, submitted for publication.

[14] J. He. Process simulation and refinement. *Journal of Formal Aspects of Computing Science*, 1:229–241, 1989.

[15] B. Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, Uppsala University, 1987. Department of Computer Systems.

[16] M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[17] Jon Kleinberg, Hagit Attiya, and Nancy Lynch. Trade-offs between message delivery and quiesce times in connection management protocols. In *Proceedings of 3rd Israel Symposium on Theory of Computing and Systems*, pages 258–267, Tel-Aviv, Israel, January 1995.

[18] Butler Lampson, Nancy Lynch, and Jørgen Søgaard-Andersen. Correctness of at-most-once message delivery protocols. In *FORTE'93 - Sixth International Conference on Formal Description Techniques*, pages 387–402, Boston, MA, October 1993.

[19] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[20] Victor Luchangco, Stephen Garland Ekrem Söylemez, and Nancy Lynch. Verificating timing properties of concurrent algorithms. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques* (FORTE'94,Berne, Switzerland, October 1994), pages 259–273. Chapman & Hall, 1995.

[21] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc, 1996.

[22] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI Quarterly*, 3(2), September 1989.

[23] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. Technical Memo MIT/LCS/TM-486, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139, May 1993.

[24] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995. Also, [23].

[25] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part II: Timing-based systems. Technical Memo MIT/LCS/TM-487.c, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, April 1995.

[26] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996. Also, [25].

[27] Sandra Murphy and A. Udaya Shankar. Connection management for the transport layer: Service specification and protocol verification. Technical Report UMIACS-TR-88-45.1, University of Maryland, June 1988. Revised December. 1989.

[28] Jon Postel. Transmission Control Protocol - DARPA Internet Program Specification (Internet Standard STC-007). Internet RFC-793, September 1981.

[29] Roberto De Prisco. Revisiting the Paxos algorithm. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1997.

[30] Editor R. Braden. Requirements for internet hosts — communication layers. Internet RFC-1122, October 1989.

[31] Roberto Segala, Rainer Gawlick, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems, 1997. Submitted for publication. An earlier version appears in [12] and a shortened version appears in [13].

[32] A. Udaya Shankar and S. S. Lam. Time-dependent distributed systems: proving safety, liveness and real-time properties. *Distributed Computing*, 2(2):61–79, 1987.

[33] A. Udaya Shankar and David Lee. Minimum-latency transport protocols with modulo-n incarnation. *IEEE/ACM Transactions on Networking*, 3(3):255–268, June 1995.

[34] J.F. Søgaard-Anderson, S.J. Garland, J.V. Guttag, N.A. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer Aided Verification. 5th International Conference, CAV '93.*, volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, June/July 1993.

[35] Jørgen Søgaard-Anderson, Nancy Lynch, and Butler Lampson. Correctness of communications protocols, a case study. Technical Report MIT/LCS/TR-589, M.I.T., November 1993.

[36] E.W. Stark. Proving entailment between conceptual state specifications. *Theoretical Computer Science*, 56:135–154, 1988.

[37] R. W. Watson. The delta-t transport protocol: Features and experience. In *IEEE 14th Conference on Local Computer Networks*, pages 399–407, October 1989.