Formal Verification of Communication Protocols

M. A. S. Smith[*]
Laboratory for Computer Science, MIT
Cambridge, MA 02139, USA, (617)253-1499, (617)253-3480, mass@lcs.mit.edu

Abstract
In this paper we present a formal abstract specification for TCP/IP transport level protocols and formally verify that TCP satisfies this specification. We also present a formal description of an experimental protocol, T/TCP, which proposes to provide the same service as TCP, but with optimizations to make it efficient for transactions. We further show that this protocol does not provide the same service as TCP, and propose a weaker specification for this protocol. 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.

Keywords
Verification, automata and languages, network protocols

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 (Braden, 1992; Braden, 1994; Braden and Clark, 1993), 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).

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.

The designers of T/TCP believed their protocol was correct since it is based on TCP,

[*]Supported by Air Force Contract AFOSR F49620-92-J-0125, NSF contract 9225124CCR, and ARPA contracts N00014-92-J-4033, F19628-95-C-0118, and DABT63-94-C-007.

but the changes they made were sufficiently complex to make them uncertain. Therefore, they thought a formal correctness proof would be useful (Braden and Clark, 1993). 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 (Søgaard-Andersen, Lampson, and Lynch, 1993) the correctness the five packet handshake protocol (Belsnes, 1976) which forms the basis of the open and close phase of TCP and ISO-TP4 is formally verified. However, this work does not verify enough of TCP for us to use directly in the verification of T/TCP. In (Murphy and Shankar, 1989) a connection management protocol for the transport layer is also specified and verified, but the protocol is of their own design, not TCP.

The informal specification of TCP (Postel, 1981) is quite complicated, and an important contribution of this work is the presentation of a precise specification of the transport level problem TCP is supposed to solve. In our formal presentation of TCP we do make some simplifications. For example, we do not include security parameters, and 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. Even with these simplifications we know of no other work that attempts to verify TCP at the level of generality we do in this work. After specifying the problem and formally verifying TCP, the next step in our verification of T/TCP was to show that it implements TCP. However, we discovered that under certain circumstances T/TCP does not behave the way TCP does, and in fact does not satisfy the specification we have for the transport layer. (Murphy, 1996) has also found a situation different from the one we discovered where T/TCP does not behave as TCP. We present the scenario we discovered in this paper and also discuss a weaker specification for T/TCP that is not violated by the scenario.

We use invariant assertion and simulation (refinement) techniques to verify TCP. We use the formalization of simulations developed in (Lynch and Vaandrager, 1993; Lynch and Vaandrager, 1995). These methods are used for proving trace inclusion relationships between concurrent systems. The methodology is developed in the context of a very simple and general automaton for both untimed (Lynch and Vaandrager, 1993) and timed (Lynch and Vaandrager, 1995) systems. We elaborate on the model and methodology in Section 2. Simulation techniques are known to be quite useful in the verification of concurrent systems, and other researchers use this method in their work (Abadi and Lamport, 1991; Lampson, Lynch, and Søgaard-Andersen 1993; Murphy and Shankar 1989). This paper is closest in scope to the work in (Lampson, Lynch, and Søgaard-Andersen, 1993).

The rest of the paper is organized as follows. Section 2 contains a brief description of the formal models we use in the paper. Section 3 contains an informal description of TCP and T/TCP. In Section 4 we present the specification of the transport layer problem, and we verify that TCP satisfies this specification in Section 5. We give a formal description of T/TCP, show how it does not behave as TCP, and discuss an alternative specification for it in Section 6. Finally, in Section 7 we make some concluding remarks and discuss future work.

2 FORMAL MODELS

In this section we give a definition of untimed and timed automata and also give a description of the simulation techniques we use.

2.1 Automata models

The formal model we use to represent the specification of the transport level problem is the untimed automaton model of (Lynch and Vaandrager, 1993). An automaton A consists of four components, a set *states(A)* of states, a nonempty set $start(A) \subseteq states(A)$ of start states, a set *acts(A)* of actions, and a set $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ of steps. The set *acts(A)* can be partitioned into three disjoint sets, *in(A)*, *out(A)*, and *int(A)* of *input actions*, *output actions* and *internal actions* respectively. The union of the *input actions* and *output actions* we denote as *external actions*, those actions visible to the environment.

We describe TCP and T/TCP as timed automata. A timed automaton (Lynch and Vaandrager, 1995) is an automaton as described above, but its set of actions includes $\mathtt{R}^+$, the set of positive reals. Actions from $\mathtt{R}^+$ are referred to as *time-passage actions*.

To show that an automaton A "implements" another automaton B we show a *trace* inclusion relationship between them. The trace inclusion relationship is a safety property and shows that a protocol does nothing bad, but does tell us if a protocol does anything (liveness). The safety property is sufficient in our work, since we are clearly dealing with protocols that do something. The set of traces of an automaton consists of the set of sequences of visible actions that the automaton can perform. In timed systems these actions are paired with their time of occurrence to form *timed traces*. Thus, A implements B if the set of (timed) traces of A is included in that of B.

2.2 Simulation techniques

In this paper we use the formalization of simulations for untimed and timed systems in (Lynch and Vaandrager, 1993; Lynch and Vaandrager, 1995) respectively. Let I be an automaton representing an implementation of a protocol and S be an automaton representing an abstract specification of the protocol. If I and S have the same input and output actions, then a simulation from I to S is a relation between states of I and states of S such that certain conditions hold. The conditions that hold depend on if we do a *forward* simulation (a special case of which is a *refinement mapping*) or a *backward* simulation. The simulation techniques have two general conditions. 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, which implies that (timed) traces of I are also (timed) traces of S. Forward and backward simulations for untimed and timed automata are shown to be sound for proving trace inclusion in (Lynch and Vaandrager, 1993; Lynch and Vaandrager, 1995).

During the process of performing a simulation proof sometimes a situation that is impossible to solve comes up, but then it turns out the situation happens in an unreachable state. Since we only need to consider the steps of the implementation automaton which start in a reachable state, an *invariant* that avoids these “bad” states is found. Invariants are properties that are true of all reachable states.

In (Abadi and Lamport, 1991) it is shown that in some instances even though it is not possible to find a refinement mapping from implementation I to specification S, by adding *history variables* to I a mapping can be found. History variables record the past history of a system and places no constraints on the behavior of the implementation.

In describing the transport level problem, it is not necessary to mention time, so our specification is presented using untimed automata. However, TCP and T/TCP are timed systems, so we presented them as timed automata. The methods we used do not allow direct simulations between timed and untimed systems. The same issue comes up in (Søgaard-Andersen, Lynch and Lampson, 1993), and they develop the *patient* operator which converts an untimed automaton into a timed automaton by adding arbitrary time passage steps.

3 INFORMAL DESCRIPTION OF PROTOCOLS

In this section we present informal descriptions of TCP and T/TCP. These description are presented here to give the reader some intuition for when we present the abstract specification and the formal descriptions of the protocols.

In order to guarantee reliable data streaming, TCP requires synchronized states at both end-points. This synchronization uses three phases: an open phase, a bi-directional data transfer phase, and a close phase. The open phase is often referred to as the “three-way handshake protocol” because it requires the sending of three packets between the client and the server. When the client and server receive the signal to open a connection, they choose initial sequence numbers (ISN), read from a 32 bit clock, from which they start numbering packets. To synchronize, the client and server must know each others ISN, so the client starts the three-way handshake by sending a SYN packet with its ISN. When the server receives this packet, it notes that the sequence number of the next packet it should receive is the ISN of the client plus one. It sends back this value to the client along with its own ISN in a SYN(ACK) packet. When the client receives this return packet, it verifies that the server received its correct ISN, and notes the ISN of the server plus one. The final packet of the three-way handshake is the packet the client sends in response. This packet has the next sequence number for the client and the value of the ISN of the server plus one. When the server receives this packet, it confirms 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. Data transfer takes place in this state.

Once the client and server agree on each other’s ISN, they increment their sequence number for each piece of data sent. The sequence numbers are used to make sure data is received in the right order. An acknowledgment mechanism is used to ensure the retransmission of packets lost in the network. That is, a packet is retransmitted after a suitable retransmission timeout (RTO) until an acknowledgment is received for that packet. The acknowledgment of a packet means every packet up to that one has been successfully received. We make the simplifying assumption that every packet must get an acknowledgment before the next one is sent.

The close phase begins when either host receives the signal to close from the user. When this happens it sends any remaining data it has to send, and then sends a FIN packet. The host that receives the FIN packet responds with a FIN(ACK) packet. The behavior is symmetric when the other host receives the signal to close. The host that sends the last FIN(ACK) goes to timed-wait state. The host that receives that FIN(ACK) closes, and the host that sent it waits for a time of $2 \times$ MPL (Maximum Packet Lifetime) before it closes. This wait is to ensure that if a new *incarnation* of the same connection is started, old duplicate packets will have been dropped from the network. Incarnations are time sequential connections of the same client/server pair. The use of the clock to choose ISN's also helps to distinguish between packets from different incarnations.

The basic idea in the design of T/TCP is to keep most of TCP, in particular the things that make it good for data streaming, but to add two optimizations that eliminates the inefficiencies of TCP for transactions. In this work we only discuss the main optimization because by itself it causes T/TCP to behave differently from TCP. The optimization, known as TCP Accelerated Open (TAO), eliminates the need for the three-way handshake protocol at the opening phase of communication. This optimization is accomplished by having *persistent* monotonic connection counts. Persistent state is state that is kept after a connection closes. Each time a new connection is opened, the connection count is incremented. The client and server hosts keep in local persistent caches their own connection counts and a copy of the last connection count they received from the host they are communicating with. Therefore, when a client wants to open a new connection, it can send the incremented connection count with the initial packet containing the request data. When the server receives this packet, it checks that the connection count is bigger that the last connection count it saw from that client, and can immediately accept the new data if it is. The server responds with a packet that contains 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. If the client or server ever lose the connection count information (for example after a crash), T/TCP uses the the three-way handshake protocol to establish connection and also to reset the connection count information. 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 0.

4 SPECIFICATION

We know of no other work that gives an abstract formal specification of the user visible behavior of TCP/IP transport level protocols. Such a specification is important because it captures the essential properties of the problem, provides precise guidelines for someone who wants to implement a transport level protocol, and provides a measure against which other transport level protocols can be checked for correctness.

Due to space limitations we present only part of the specification here. The full specification appears in (Smith, 1996). 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 specification, S, is shown in Figure 1.

The user interface for TCP in the Internet standard (Postel, 1981), has an explicit *active-open* input and separate *send-msg* and *close* inputs. We combined these actions in our specification into the single $\textit{send-msg}_c\textit{(open, m, close)}$ * action on the client side because we want to 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 (Braden, 1994) 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.

We use the untimed automaton model described in Section 2 to formally present the specification. The steps are presented in a *precondition, effect* style commonly used with I/O automata (Lynch and Tuttle, 1989). 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. Input actions have no precondition.

Some of the steps† of the automaton for the specification, S, are shown in Figure 2. 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 variables rec_c and rec_s are used to indicate when the client and server respectively are recovering from crashes. Most actions are disabled or have no effect when these variables are true. To represent the idea of the sides opening and forming a connection, we assign id's from infinite and stable sets (*CID* and *SID* for the client and server side respectively) to the client and the server ends when they open, and then pair them to form an association. A stable variable is one that does not lose its value after a crash. To make sure associations are distinct, id's are removed from the sets once they are used, and an id is never paired with more than one other id to form an association. We use another stable set, *assoc*, to keep track of the associations that have already been formed, and the special value `nil` is used to indicate when a host is closed.

To ensure the separation of data for each incarnation, we use two infinite arrays of FIFO queues that are indexed by the id's of the client and server. That is, the client sends data

---

* *open* and *close* are boolean, and $m \in \text{Msg} \cup \texttt{null}$, 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` which indicates the absence of a message.

† Throughout the paper we use the following operations on queues: let q be a queue $\langle e_0, \ldots, e_{n-1}\rangle$ with elements e_0 through e_{n-1}. Then $q \cdot m$ and tail(q) denote the lists $\langle e_0, \ldots, e_{n-1}, m\rangle$, and $\langle e_1, \ldots, e_{n-1}\rangle$, respectively, and head$(q)$ is the element e_0. Also let $\textit{dom(q)} \triangleq \{i|0 \le i < |q|\}$, $\textit{suffix(q)} \triangleq \{\{i|j \le i < |q|\}|0 \le j < |q|\}$, and $\textit{delete(q,I)} \triangleq \langle q|i| \mid i \in \textit{dom(q)} \wedge i \notin I\rangle$. The empty queue is denoted by ϵ.

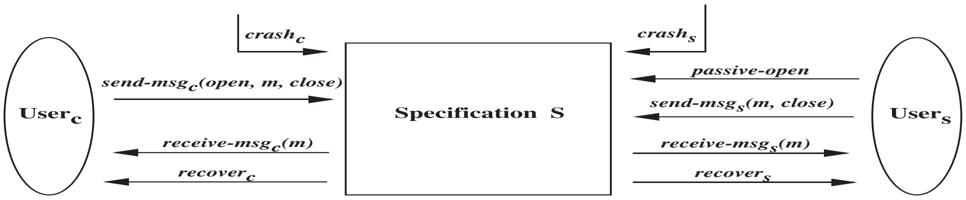

Figure 1 The user interface for TCP/IP transport level protocols.

on the queue indexed by its id ($queue_{sc}$), and the server sends data on the queue indexed by its id ($queue_{sc}$). A host can only receive data (the *receive-msg*$_c(m)$ *and receive-msg*$_s(m)$ actions) 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-stat*$_{cs}$ and *q-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.

When a host opens it also sets the variable *mode* to `active` to indicate that it can receive messages from the user, and sets it to `inactive` when it receives the close signal from the user. A host should only close, barring a crash, when it has sent all its data (it received a *close* signal from the user) and when it has received all the data from the other host. In the specification a remote host can determine when the other host has sent all its data by checking the *mode* or *id* variable of the other host. The internal action to close a host is *set-nil*.

Our specification includes another stable set, *overlap*, which contains all the pairs of id's of clients and servers that co-existed in time. We need this set because we want to allow associations to be formed by id's that are not necessarily the current id's of the client and server, but id's that existed at the same time.

5 TCP

We specify the client and server as timed automata. The protocol is too large to present in its entirety in this paper, so we only present the steps of the open phase. The steps are also somewhat simplified to ease the exposition in this paper. For the full protocol see (Smith, 1996).

Figure 3 shows the structure of the TCP protocol, T. We use two channels, Channel$_{cs}$ and Channel$_{sc}$, for sending packets from the client to the server and from the server to the client respectively. We model the channels as timed automata. When a packet gets placed on the network and is not dropped, it eventually gets delivered. However packets can be lost, duplicated and reordered. A packet that is placed on a channel and does not get delivered within the MPL gets dropped from the channel. In our model packets on the channels are stored in the multiset variables *in-transit*$_{cs}$ and *in-transit*$_{sc}$ for packets on Channel$_{cs}$ and Channel$_{sc}$ respectively.

Client side

Input *send-msg$_c$(open, m, close)*
Effect:
if $\neg\, rec_c$ then
if $open \wedge id_c = \texttt{nil}$ then
$id_c :=$ any element $\in$ CID
CID := CID $\setminus\, id_c$
$mode_c := \texttt{active}$
$q\text{-}stat_{cs}(id_c) := \texttt{live}$
if $id_s \neq \texttt{nil}$ then
$overlap := overlap \cup \{(id_c, id_s)\}$
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 $mode_c := \texttt{inactive}$

Internal *make-assc(i,j)*
Precondition: $(i, j) \in overlap \wedge$
$\forall\, k(i, k) \notin assoc$
$\wedge\, \forall\, l\ (l, j) \notin assoc$
Effect: $assoc := assoc \cup \{(i, j)\}$

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

Internal *set-nil$_c$*
Precondition: $\neg\, rec_c \wedge$
$id_c \neq \texttt{nil} \wedge mode_c = \texttt{inactive}$
$\wedge\, \exists j$ s.t. $(id_c, j) \in assoc \wedge queue_{sc}(j) = \epsilon$
$\wedge\, (mode_s = \texttt{inactive} \vee id_s \neq j)$
Effect:
$id_c := \texttt{nil}$
$q\text{-}stat_{sc}(j) := \texttt{dead}$

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

Internal *lose$_c$(I)*
Precondition: $rec_c \wedge I \in suffix(queue_{cs}(id_c))$
Effect: $queue_{cs}(id_c) := \text{delete}(queue_{cs}(id_c), I)$

Server side

Input *passive-open*
Effect:
if $\neg\, rec_s$ then
if $id_s = \texttt{nil}$ then
$id_s :=$ any element $\in SID$
$SID := SID \setminus id_s$
$mode_s := \texttt{active}$
$q\text{-}stat_{sc}(id_s) := \texttt{live}$
if $id_c \neq \texttt{nil}$ then
$overlap := overlap \cup \{(id_c, id_s)\}$

Input *send-msg$_s$(m, close)*
Effect:
if $\neg\, rec_s$ 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 $mode_s := \texttt{inactive}$

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

Internal *set-nil$_s$*
Precondition: $\neg\, rec_s \wedge$
$id_s \neq \texttt{nil} \wedge mode_s = \texttt{inactive} \wedge$
$\exists i$ s.t. $(i, id_s) \in assoc \wedge queue_{cs}(i) = \epsilon$
$\wedge\, (mode_c = \texttt{inactive} \vee id_c \neq i)$
Effect:
$id_s := \texttt{nil}$
$q\text{-}stat_{cs}(i) := \texttt{dead}$

Input *crash$_s$*
Effect:
if $id_s \neq \texttt{nil}$ then
$\text{rec}_s := \texttt{true}$
if $\exists\, i$ s.t. $(i, id_s) \in assoc$ then
$queue_{cs}(i) := \epsilon$
$q\text{-}stat_{cs}(i) := \texttt{dead}$

Internal *lose$_s$(I)*
Precondition: $rec_s \wedge I \in suffix(queue_{sc}(id_s))$
Effect: $queue_{sc}(id_s) := \text{delete}(queue_{sc}(id_s), I)$

Figure 2 Some of the steps of the specification S.

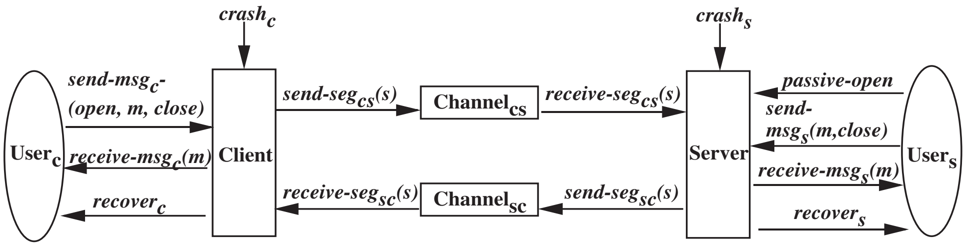

Figure 3 The Structure of TCP.

5.1 Steps of TCP

The steps of the open phase of the timed automaton for TCP are shown in Figure 4. The protocol starts when the client receives the action *send-msg*$_c$*(open, m, close)* with *open* set to true and the server receives a *passive-open* 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. When the client and server receive the open signal they change to modes `syn-sent` and `listen` respectively. These modes indicate that the client is about to or has sent a SYN packet and that the server is listening for one. We represent the choosing of the ISN's as the routines *choose-isn*$_c$*()* and *choose-isn*$_s$*()*. 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.

Assuming the client does not open and close without receiving any data, the client performs the action *send-seg*$_{cs}$*(SYN, sn*$_c$*)*, 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* actions have as a precondition ($now_c - \textit{time-sent}_c \geq \text{RTO}$) which controls retransmission. When this packet is received by the server, if it is in mode `listen`, it changes to mode `syn-rcvd` and also records the next sequence number it expects, sn_c+1, in the variable ack_s. After it receives the first packet of the three-way handshake, the server performs the action *send-seg*$_{sc}$*(SYN, sn*$_s$*, ack*$_s$*)* which is the second packet of the three-way handshake. In this packet sn_s is the server's ISN. When the client receives this packet, it accepts it only if it is in *mode* `syn-sent` and it knows the server received its correct ISN. The client knows the server received its correct ISN if $ack_s = sn_c + 1$. If the received packet is valid, the client goes to *mode* `estb` and makes assignments in preparation for sending the final packet in the three-way handshake. First ack_c is set to $sn_s + 1$ for the next expected packet, and *time-sent*$_c$ gets set to 0. Then if there is data to be sent, the flag *prep-msg*$_c$ is set to enable the internal action, *prepare-msg*$_c$. If there is no data to be sent, then the sending of just an acknowledgment is enabled by setting *send-ack*$_c$. If there is data in *send-buf*$_c$, the *prepare-msg*$_c$ 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 will get sent with the next packet. The final part of the 3-way-handshake is the action *send-seg*$_{cs}$*(sn*$_c$*, ack*$_c$*, msg*$_c$*)* that acknowledges the SYN packet from the server. The precondition of this action is to prevent the sending of a final ack before all received data has been passed to the user, and the condition in the effect clause sets up the wait of timed-wait state if the ack is a FIN(ACK). In the

Client side

Input $send\text{-}msg_c(open,\ m,\ close)$
Effect: if $mode_c = \texttt{closed} \wedge open$ then
  $sn_c := choose\text{-}isn_c()$
  $mode_c := \texttt{syn-sent}$
 if $mode_c \in \{\texttt{syn-sent}, \texttt{estb}, \texttt{close-wait}\} \wedge$
 $\neg\ rcvd\text{-}close_c \wedge\ m\ \neq \texttt{null}$ 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
   $mode_c := \texttt{closed}$

Internal $send\text{-}seg_{cs}(SYN,\ sn_c)$
Precondition: $(now_c$ - $time\text{-}sent_c \geq \mathrm{RTO}) \wedge$
 $mode_c = \texttt{syn-sent}$
Effect: $time\text{-}sent_c := now_c$

Internal $receive\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$
Effect: if $mode_c = \texttt{syn-sent} \wedge ack_s = sn_c{+}1$ 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 $prep\text{-}msg_c := \texttt{true}$
  else $send\text{-}ack_c := \texttt{true}$

Internal $prepare\text{-}msg_c$
Precondition: $mode_c \neq \texttt{rec} \wedge prep\text{-}msg_c \wedge$
 $\vee\ (mode_c \in \{\texttt{estb}, \texttt{close-wait}\} \wedge\ \neg ready\text{-}to\text{-}send_c$
 $\wedge\ (send\text{-}buf_c \neq \epsilon\ \vee rcvd\text{-}close_c))$
Effect: $prep\text{-}msg_c := \texttt{false}$
 $ready\text{-}to\text{-}send_c := \texttt{true}$
 if $send\text{-}buf_c \neq \epsilon$ then
  $sn_c := sn_c + 1$
  $msg_c := \mathrm{head}(send\text{-}buf_c)$
  $send\text{-}buf_c := \mathrm{tail}(send\text{-}buf_c)$
 if $rcvd\text{-}close_c \wedge send\text{-}buf_c = \epsilon$ then
  $sn_c := sn_c + 1$
  $ready\text{-}to\text{-}send_c := \texttt{false}$
  $send\text{-}fin_c := \texttt{true}$
  if $mode_c = \texttt{estb}$ then $mode_c := \texttt{fin-wait-1}$
  else if $mode_c = \texttt{close-wait}$ then
   $mode_c := \texttt{last-ack}$

Internal $send\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$
Precondition: $(now_c$ - $time\text{-}sent_c \geq \mathrm{RTO}) \wedge$
 $((ready\text{-}to\text{-}send_c \vee send\text{-}ack_c) \wedge$
 $mode_c \in \{\texttt{estb}, \texttt{fin-wait-1}, \texttt{fin-wait-2}\}) \vee$
 $((ready\text{-}to\text{-}send_c \vee send\text{-}ack_c) \wedge$
 $mode_c \in \{\texttt{closing}, \texttt{timed-wait}\ \texttt{close-wait}\}$
 $\wedge\ rcv\text{-}buf_c = \epsilon)$
Effect: $time\text{-}sent_c := \mathrm{now}_c$
 if $mode_c = \texttt{timed-wait}$ then
  $start\text{-}wait_c := now_c$
  $time\text{-}sent_c := \infty$

Server side

Input $passive\text{-}open$
Effect: if $mode_s = \texttt{closed}$ then
  $sn_s := chose\text{-}isn_s()$
  $mode_s := \texttt{listen}$

Input $send\text{-}msg_s(m,\ close)$
Effect: if $mode_s \in \{\texttt{syn-rcvd}, \texttt{estb}, \texttt{close-wait}\} \wedge$
 $\neg\ rcvd\text{-}close_s \wedge m\ \neq \texttt{null}$ 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
   $mode_s := \texttt{closed}$

Internal $receive\text{-}seg_{cs}\ (SYN,\ sn_c)$
Effect: if $mode_s = \texttt{listen}$ then
  $mode_s := \texttt{syn-rcvd}$
  $ack_s := sn_c + 1$
  $time\text{-}sent_s := 0$

Internal $send\text{-}seg_{sc}(SYN,\ sn_s,\ ack_s)$
Precondition: $(now_s$ - $time\text{-}sent_s \geq \mathrm{RTO}) \wedge$
 $mode_s = \texttt{syn-rcvd}$
Effect: $time\text{-}sent_s := now_s$

Internal $prepare\text{-}msg_s$
Precondition: $mode_s \neq \texttt{rec} \wedge prep\text{-}msg_s \wedge$
 $\vee\ (mode_s \in \{\texttt{estb}, \texttt{close-wait}\}\ \wedge\ \neg ready\text{-}to\text{-}send_s$
 $\wedge\ (send\text{-}buf_s \neq \epsilon \vee rcvd\text{-}close_s))$
Effect: $prep\text{-}msg_s := \texttt{false}$
 $ready\text{-}to\text{-}send_s := \texttt{true}$
 if $send\text{-}buf_s \neq \epsilon$ then
  $sn_s := sn_s + 1$
  $msg_s := \mathrm{head}(send\text{-}buf_s)$
  $send\text{-}buf_s := \mathrm{tail}(send\text{-}buf_s)$
 if $rcvd\text{-}close_s \wedge send\text{-}buf_s = \epsilon$ then
  $sn_s := sn_s + 1$
  $ready\text{-}to\text{-}send_s := \texttt{false}$
  $send\text{-}fin_s := \texttt{true}$
  if $mode_s = \texttt{estb}$ then $mode_s := \texttt{fin-wait-1}$
  else $mode_s = \texttt{close-wait}$ then
   $mode_s := \texttt{last-ack}$

Internal $receive\text{-}seg_{cs}(sn_c,\ ack_c,\ msg_c)$
Effect: if $mode_s \neq \texttt{rec}$ then
  if $sn_c = ack_s$ then
   $ack_s := sn_c + 1$
   $time\text{-}sent_s := 0$
   $rcv\text{-}buf_s := rcv\text{-}buf_s \cdot msg_c$
   $send\text{-}ack_s := \texttt{true}$
  if $ack_c = sn_s + 1$ then
   $ready\text{-}to\text{-}send_s := \texttt{false}$
   $send\text{-}fin_s := \texttt{false}$
   if $mode_s = \texttt{syn-rcvd}$ then $mode_s := \texttt{estb}$
   if $send\text{-}buf_s \neq \epsilon$ then
    $prep\text{-}msg_s := \texttt{true}$
    $send\text{-}ack_s := \texttt{false}$

Figure 4 Steps from the open phase of TCP.

open phase, when the server receives the corresponding input, $mode_s$ will be `syn-rcvd` and it will then change to `estb`. If there is valid data in the packet, that is $sn_c = ack_s$, it is placed on the receive buffer, ack_s is incremented, and $send\text{-}ack_s$ is set to true.

5.2 The correctness of TCP

In S, we can only lose messages between a crash and a recovery. In some low-level protocols, whether a message gets lost or not may not be decided until after recovery. This decision is dependent on *race conditions* that may exist on the channels. The postponing of nondeterministic choices in the implementations suggests the need for a backward simulation. A similar situation comes up in (Søgaard-Andersen, Lynch and Lampson, 1993) and they develop the idea of a Delayed-Decision Specification. We use this idea, and our Delayed-Decision Specification D is similar to the one in their work. The main idea of a Delayed Decision Specification is to have it as an intermediate specification that has the postponed nondeterminism we see in implementations. This idea allows us to do a backward simulation from D to S instead of doing it directly from the implementation to S, and then a refinement mapping from the implementation to D. Doing the simulations in this manner is useful 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 the two simulations turns out to be easier than a direct backward simulation.

In D messages on the queues are tagged with either `ok` or `marked`. Also, instead of deleting messages between a crash and a recovery, D marks these messages. Marking changes an `ok` tag to `marked`. Marked messages can then be dropped at any time, but because only marked messages can be dropped, only messages that were in the system at the time of a crash can be deleted. Marked messages can still be delivered to users. The specification D and the backward simulation from D to S are presented in (Smith, 1996).

The next step in the verification of TCP is to show the refinement mapping from automaton T to D. As we discussed in Section 2 we have to apply the *patient* operator to D to get a timed version. We call this new automaton D^p. The *Embedding Theorem* of (Søgaard-Andersen, Lynch and Lampson, 1993) states that automaton A implements an automaton B iff *patient(A)* implements *patient(B)*. Applying the theorem here means D^p implements *patient(S)*. We also add history variables to T. We call the resulting automaton T^h. The history variables we add to T are *assoc*, *overlap*, isn_c, and isn_s; *assoc* and *overlap* correspond to the variables of the same name in the specification, and isn_c and isn_s correspond to the id's of an incarnation on the client and the server side respectively. We also need some invariants on the reachable states of T^h. These invariants, I_T, are presented and proved in (Smith, 1996).

In our definition of the refinement mapping from T^h to D^p we write *k.variable* to denote the value of *variable* in state k. We also use the notation $s.current\text{-}msg_c$ and $s.pos\text{-}pac_c$ to represent the *current message* and a *possible packet* respectively on the client side in state s. The *current message* is the message that is being sent, but has not yet been received paired with the value `ok` to match variables on the queues in specification D. When the message is received, $current\text{-}msg_c$ becomes the empty string since the particular message will be on the receive buffer. A *possible packet* is a set consisting of packets with a message that could still possibly be delivered even though the sender crashes before receiving an

1.	$u.id_c$	$=$	$s.isn_c$	if $s.mode_c \neq \texttt{closed}$
		$=$	$\texttt{nil}$	if $s.mode_c = \texttt{closed}$
2.	$u.mode_c$	$=$	$\texttt{active}$	if $s.mode_c \in \{\texttt{syn-sent}, \texttt{estb}, \texttt{close-wait}\} \wedge \neg\, s.rcvd\text{-}close_c$
		$=$	$\texttt{inactive}$	if $s.rcvd\text{-}close_c \vee mode_c = \texttt{closed}$
3.	$u.q\text{-}stat_{cs}(i)$	$=$	$\texttt{live}$	if $(s.isn_c = i \wedge (i, s.isn_s) \notin s.assoc) \vee ((i, s.isn_s) \in s.assoc \wedge s.mode_s \notin \{\texttt{rec}, \texttt{closed}\})$
		$=$	$\texttt{dead}$	otherwise
4.	$u.queue_{cs}(i)$	$=$	ϵ	if $\neg(s.isn_c = i \vee (i, s.isn_s) \in s.assoc) \vee (s.mode_s \in \{\texttt{rec}, \texttt{closed}\} \wedge (i, s.isn_s) \in s.assoc)$ **(A)**
		$=$	$(s.send\text{-}buf_c \times \texttt{ok})$	if $s.isn_c = i \wedge s.mode_c \in \{\texttt{syn-sent}, \texttt{rec}\} \wedge (i, s.isn_s) \notin s.assoc$ **(B)**
		$=$	concatenation of: • $(s.rcv\text{-}buf_s \times \texttt{ok})$ • $(msg(s.pos\text{-}pac_c) \times \texttt{marked})$	if $((s.isn_c \neq i \wedge (i, s.isn_s) \in s.assoc) \vee (s.mode_c = \texttt{rec} \wedge i = s.isn_c \wedge (i, s.isn_s) \in s.assoc)) \wedge s.mode_s \notin \{\texttt{rec}, \texttt{closed}\}$ **(C)**
		$=$	concatenation of: • $(s.rcv\text{-}buf_s \times \texttt{ok})$ • $s.current.msg_c$ • $(s.send\text{-}buf_c \times \texttt{ok})$	if $i = s.isn_c \wedge (s.mode_c = \texttt{estb} \vee (i, s.isn_s) \in s.assoc) \wedge s.mode_c \neq \texttt{rec} \wedge s.mode_s \notin \{\texttt{rec}, \texttt{closed}\}$ **(D)**

Figure 5 Part of the refinement mapping R_{TD} from T^h to D^p.

acknowledgment of this packet. The packet may or may not get delivered depending on if all copies get dropped from the channel or if the receiving side crashes. The message by the crashed sender may still be delivered while the sender is recovering or is in an unsynchronized state, but is definitely lost if the crashed sender recovers and gets back to a synchronized state. This is the case because after a side 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 the other side closed and reopened, thus starting a new incarnation. We use the notation *msg(packet)* to mean the message from a packet, so for example, $msg(s.pos\text{-}pac_c)$ is $s.msg_c$.

We define a function R_{TD} from *states*(T^h) to *states*(D^p). In the definition, when we write, for example, "$(send\text{-}buf_c \times \texttt{ok})$", we mean the element of $(Msg \times \texttt{ok})^*$ obtained from $send\text{-}buf_c$ by pairing every message with $\texttt{ok}$. The main definitions for R_{TD} are shown in Figure 5. We only show the mappings for client side variables since the mappings for server side variables are basically symmetric. If $s \in$ *states*(T^h) then $R_{TD}(s)$ is the state $u \in$ *states*(D^p) such that the equations of R_{TD} hold.

The intuition behind the mappings of $u.id_c$ and $u.mode_c$ is straightforward. The more difficult cases are for $u.q\text{-}stat_{cs}(i)$ and $u.queue_{cs}(i)$. In T^h there are four variables that possibly correspond to parts of the abstract queues in D^p for messages going from client to server. These variables are $s.send\text{-}buf_c$, $s.msg_c$, $s.in\text{-}transit_{cs}$, and $s.rcv\text{-}buf_s$. As soon as the client opens and assigns $s.isn_c$, the corresponding queue is activated, so $u.q\text{-}stat_{cs}(s.isn_c)$ becomes $\texttt{live}$. In this situation, $(s.send\text{-}buf_c \times \texttt{ok})$ (case **(B)**) corresponds to the abstract queue. If the client opens right after it crashed, the server side might still be receiving data from the previous incarnation, that is, if $(i, s.isn_s) \in assoc \wedge i \neq s.isn_c$, so that queue may still be `live`. In that case the message from the possible packet on the channel paired with `marked`, since it might get dropped, concatenated with $(s.rbuf_s \times$ `ok`$)$, that is queues from **(C)**, correspond to the abstract queue. The status of a queue is also `live` when both client and server are up and their *isn*'s have formed an association. Queues in group **(D)** correspond to this situation. The tricky part of this mapping is dealing with the current message being sent, $s.msg_c$, because it may have duplicates on the channel and another duplicate in the receive buffer of the server. Our definition of $s.current\text{-}msg_c$ handles this by becoming the empty string when $s.msg_c$ is received at the server, and the duplicates on the channel are ignored until there is a crash, in which case $u.queue_{cs}(i)$ is in group **(C)**. A queue that is `live` becomes `dead` when its receiving end stops accepting data because of a close or a crash.

Lemma 1 *R_{TD} is a refinement mapping from T^h to D^p with respect to I_T.*

Proof. The proof of this Lemma is in (Smith, 1996). We give a brief sketch here. The proof has two parts. The first part, which is quite straightforward, is to show that the start states of T^h and D^p correspond. The second part is to show that for each step (s, a, s') of T^h, where s and s' satisfy the invariant I_T, there exists a sequence of steps $(u, a_1, \ldots, a_n, u')$ of D^p with the same timed trace. This part is done by doing case analysis for each step, a, of T^h and is quite long. ∎

Theorem 1 *TCP implements the patient version of specification S.*

Proof. This Theorem follows from the *Embedding Theorem*, Lemma 1 and the soundness of backward simulations and refinement mappings. ∎

6 T/TCP

In this Section we show some of the steps of the timed automaton for T/TCP. We also present the situation where T/TCP does not behave like TCP and propose an alternative specification that would allow this situation.

6.1 Steps of T/TCP

The steps shown in Figure 6 are the ones necessary for a transaction using TAO. When the client opens it increments its connection count (cc_gen_c) and assigns that value to cc_send_c which stores the connection count value the client will send for the current incarnation of the connection. If $cache\_cc\_sent_c$ (the persistent copy of the connection count for the previous incarnation) is undefined or greater that cc_send_c it means that this value cannot be used. In this case a three-way handshake is needed. Those steps are not shown here. Otherwise the client prepares to send a packet with the *CC* option which means a TAO test should be done by the server when the packet is received. If there is a message, it is placed on $send\text{-}buf_c$ and $prep\text{-}msg_c$ is set to true to enable the internal action $prepare\text{-}msg_c$ which is not shown here, but is very similar to the one for TCP in Figure 4.

Client side

Input *send-msg*$_c$*(open, m, close)*
Effect: if $mode_c$ = `closed` $\wedge$ *open* then
    $cc\_gen_c := cc\_gen_c + 1$
    $cc\_send_c := cc\_gen_c$
    $sn_c := 0$
    $mode_c :=$ `syn-sent`
    if $cache\_cc\_sent_c$ is undefined $\vee$
    $cc\_send_c < cache\_cc\_sent_c$ then
        $cc\_new_c :=$ `true`
        $cache\_cc\_sent_c := 0$
    else
        $cc_c :=$ `true`
        $cache\_cc\_sent_c := cc\_send_c$
  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` $\wedge$ $cc_c \wedge sn_c = 0$ then
        $prep\text{-}msg_c :=$ `true`
  if *close* then
    $rcvd\text{-}close_c :=$ `true`
    if $mode_c$ = `syn-sent` $\wedge$ $sn_c = 0$
    $\wedge$ $send\text{-}buf_c = \epsilon$ then
        $mode_c :=$ `closed`

Internal $send\text{-}seg_{cs}$ –
(SYN, CC, cc_send_c, sn_c, msg_c, *FIN)*
Precondition: (now_c - $time\text{-}sent_c \geq$ RTO) $\wedge$
  $mode_c$ = `syn-sent*` $\wedge$ $cc \wedge \neg$ $send\text{-}rst_c \wedge send\text{-}fin_c$
Effect: $time\text{-}sent_c := now_c$

Internal $receive\text{-}seg_{sc}$ –
(SYN, CCE, cc_rcvd_s, sn_s, ack_s, msg_s, *FIN)*
Effect: if $cc\_rcvd_s = cc\_send_c$ then
    if $mode_c$ = `syn-sent` then
        $mode_c :=$ `close-wait`
    if $mode_c$ = `syn-sent*` then
        $mode_c :=$ `timed-wait`
    $cache\_cc\_sent_c := cc\_send_c$
    $time\text{-}sent_c := 0$
    $ready\text{-}to\text{-}send_c :=$ `false`
    $send\text{-}fin_c :=$ `false`
    if $sn_s = ack_c$ then
        $rcv\text{-}buf_c := rcv\text{-}buf_c \cdot m$
        $ack_c := sn_s + 1$
        $send\text{-}ack_c :=$ `true`
    if $send\text{-}buf_c \neq \epsilon$ then
        $prep\text{-}msg_c :=$ `true`
        $send\text{-}ack_c :=$ `false`

Server side

Input *passive-open*
Effect: if $mode_s$ = `closed` then
    $cc\_send_s := \mathrm{inc}(cc\_gen_s)$
    $sn_s := 0$
    $mode_s :=$ `listen`

Input *send-msg*$_s$*(m, close)*
Effect:
  if $mode_s \in \{$`syn-rcvd`, `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`

Internal $receive\text{-}seg_{cs}$ –
(SYN, CC, cc_send_c, sn_c, msg_c, *FIN)*
Effect: if $mode_s$ = `listen` then
    $cc\_rcvd_s := cc\_send_c$
    $ack_s := sn_c + 1$
    $time\text{-}sent_s := 0$
    if $cache\_cc_s > cc\_send_c$ then
        $cache\_cc_s := cc\_send_c$
        $rcv\text{-}buf_s := rcv\text{-}buf_s \cdot m$
        $cc\_echo :=$ `true`
        $mode_s :=$ `close-wait*`
    else
        $mode_s :=$ `syn-rcvd`
        $cache\_cc_s := 0$
        $temp\text{-}data := m$
        $fin\text{-}rcvd :=$ `true`

Internal $send\text{-}seg_{sc}$ –
(SYN, CCE, cc_rcvd_s, sn_s, ack_s, msg_s, *FIN)*
Precondition: (now_s - $time\text{-}sent_s \geq$ RTO) $\wedge$
  $mode_s \in \{$`fin-wait1*`, `last-ack*`$\}$ $\wedge$
  $cc\_echo \wedge send\text{-}fin_s$
Effect: $time\text{-}sent_s := now_s$

Figure 6 Steps for T/TCP Accelerated Open.

The next action shown for the client, *send-seg*$_{cs}$*(SYN, CC, cc_send*$_c$*, sn*$_c$*, msg*$_c$*, FIN)*, is enabled if the client when it opened got a message to send, and also the signal to close. This is the type of transaction T/TCP is meant to optimize. When the server receives this packet, *receive-seg*$_{cs}$ *(SYN, CC, cc_send*$_c$*, sn*$_c$*, msg*$_c$*, FIN)*, it checks if the value of *cc_send*$_c$ is greater than the connection count value of the previous incarnation (*cache_cc*$_s$). If it is, the TAO test is passed and the data can be accepted, otherwise the data is stored in a temporary variable until it can be validated by a three-way handshake. After the server passes the data to the user, if it gets a *send-msg*$_s$*(m, close)* input with response data and the signal to close from the user, it sends the packet with the response data to the client with the action *send-seg*$_{sc}$*(SYN, CCE, cc_rcvd*$_s$*, sn*$_s$*, ack*$_s$*, msg*$_s$*, FIN)*. When the client receives this packet, it checks that the echoed connection count value, *cc_rcvd*$_s$, is the same connection count value it sent. If the data is valid, the client can pass it to the user on its side, which means a transaction is performed in only one round trip across the network.

6.2 T/TCP behaves differently from TCP

The situation where the TAO mechanism cause T/TCP to behave differently from TCP occurs in the execution described above, if the server crashed after it passed the data to the user, but before it had a chance to send a response, and then after it recovers and reopens, it receives a new copy of the initial packet (either a retransmission or a duplicate from the network). This reception causes it to go through the three-way handshake protocol and accept the message again. The three-way handshake is necessary because *cache_cc*$_s$ is undefined after a crash. This delivery of the same data twice does not happen in TCP, and also violates our specification. The duplicate delivery occurs because neither the client nor server can tell that the message had been delivered before.

The designers of T/TCP do not seem to think that this behavior of T/TCP is necessarily bad. Therefore, a weaker specification must be formulated that allows this behavior. The key observation is that T/TCP has a weaker notion of an association. That is, in T/TCP an id chosen by either side can form an association with more than one id from the other side. However, we want to allow an id that is already in the set of associations to form an association with another id only if the previous id's it is associated with are *crash id's*. Crash id's are id's a host has when it crashes. To incorporate this idea into our specification, we need variables to keep track of the crash id's on both sides, and we change the precondition of *make-assoc(i,j)* to allow associations to be formed if neither i nor j are currently in the set *assoc*, or if one of them is there already, but the id's it is paired with are in the set of crash id's.

7 CONCLUSION

In this paper we presented a formal abstract specification, using a simple automaton model, for the problem of reliable data delivery for transport level protocols. The specification gives the precise requirements for these protocols without the clutter of implementation details. It is the first such specification for the user visible behavior of TCP, and it can be used to guide the design of other transport level protocols. Using a timed version of the automaton model, we also presented TCP, and formally verified that it satisfied the specification of the problem. Our formal verification of TCP is the most comprehensive that we know of to date, and is further indication that the models and techniques used in this work are viable for verifying large practical protocols. We also presented the experimental protocol T/TCP. The verification of this protocol was the original motivation for our work. The designers of the protocol thought it implemented TCP; however, we have shown that it does not. The behavior that T/TCP exhibits might still be acceptable for some applications and we proposed a weaker specification for T/TCP.

Currently we are working to formally verify that T/TCP satisfies our weaker specification. We would also like to formulate precisely the conditions under which T/TCP does implement TCP, which will give insights into which applications T/TCP can be substituted for TCP without the potential for problems. Finally, we are also interested in either designing a protocol that satisfies our initial specification and can still perform a transaction in one round trip across the network, or formally showing that it is impossible to design such a protocol.

REFERENCES

Abadi, M. and Lamport, L. (1991) The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284.

Belsnes, D. (1976) Single message communication. *IEEE Transactions on Communications*, 24(2).

Braden, R. (1992) Extending TCP for transactions – concepts. Internet RFC-1379.

Braden, R. (1994) T/TCP – TCP extensions for transactions – functional specification. Internet RFC-1644.

Braden, R. and Clark, D. (1993) Transport protocols for transactions and streaming. Unpublished manuscript.

Lampson, B., Lynch N, and Søgaard-Andersen, J. (1993) Correctness of at-most-once message delivery protocols. In *FORTE'93 - Sixth International Conference on Formal Description Techniques*, pages 387–402, Boston, MA.

Lynch, N. and Tuttle, M. (1989) An introduction to input/output automata. *CWI Quarterly*, 3(2).

Lynch, N. and Vaandrager, F. (1995) Forward and backward simulations - part II: Time-based systems. To Appear in Information and Computation.

Lynch, N. and Vaandrager, F. (1993) Forward and backward simulations - part I: Untimed systems. Technical Memo MIT/LCS/TM-486, M.I.T..

Murphy, S. (1996) Private communication.

Murphy, S. and Shankar, A. (1989) 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.

Postel, J. (1981) Transmission conrol protocol - DARPA Internet program specification (Internet standard STC-007). Internet RFC-793.

Søgaard-Anderson, J., Lynch, N. and Lampson, B. (1993) Correctness of communications protocols, a case study. Technical Report MIT/LCS/TR-589, M.I.T..

Smith, M. (1996) *Formal Verification of Communications Protocols for Data Streaming and Transactions*. PhD thesis, M.I.T. In progress.