Specifying and Using a Partitionable Group Communication Service

DRAFT - Do Not Distribute

Alan Fekete* Nancy Lynch† Alex Shvartsman‡

October 17, 1997

Abstract

Group communication services are becoming widely accepted as useful building blocks for the construction of fault-tolerant distributed applications. Many system designers and researchers have proposed formal specifications for group communication services. However, there is still no agreement about what these specifications should say, especially in cases where the services are *partitionable*, that is, where disjoint sets of members may simultaneously believe they constitute the whole group.

In this paper, we present a new, simple specification for a partitionable group communication service. We use the specification to construct an ordered-broadcast application, using an algorithm (based on algorithms of Amir, Dolev, Keidar and others) that reconciles information derived from different instantiations of the group. We prove the correctness and analyze the performance and fault-tolerance of the resulting application. Our specification has a simple implementation, based on the membership algorithm of Cristian and Schmuck.

The service we specify associates each message with a particular *view* of the group membership. All send and receive events for a message occur at processors when they have the associated view. The service provides a total order on the messages within each view, and each processor receives a prefix of this total order.

The style of our specification is different from those of previous specifications for group communication services, in that we separate safety requirements from performance and fault-tolerance requirements. The safety requirements are expressed by an abstract, global *state machine*. To present the performance and fault-tolerance requirements, we include *failure-status input actions* in the specification; we then give properties saying that consensus on the view and timely message delivery are guaranteed in an execution provided that the execution *stabilizes* to a situation in which the failure status stops changing and corresponds to a consistently partitioned system. (This stabilization hypothesis can be seen as an abstract version of the "timed asynchronous model" of Cristian.)

Our specification is simple; for example, it does not mention any "hidden" or "transitional" views as do some others. It is also significantly weaker than most others, for example, we do not require consensus on either knowledge of group membership or on message delivery, except in those cases where the execution stabilizes. Because consensus is not required in every execution, the specification is not subject to the existing impossibility results for partitionable systems.

Despite its weakness, the specification is sufficient to allow important applications to run on top of the group communication service. In this paper, we present one such application, the ordered-broadcast application mentioned above. The application manages the view-change activity to build a shared sequence of messages, that is, the per-view total orders of the group service are combined to give a universal total order. The ordered-broadcast application can itself be used to implement sequentially consistent memory,

*Basser Department of Computer Science, Madsen Building F09, University of Sydney, NSW 2006, Australia. Email: `fekete@cs.usyd.edu.au`

†Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, NE43-365, Cambridge, MA 02139, USA. Email: `lynch@theory.lcs.mit.edu`.

‡Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, NE43-371, Cambridge, MA 02139, USA. Email: `alex@theory.lcs.mit.edu`.

using simple replicated data management. Our state-based specification is used in an assertional proof of the safety properties of the ordered-broadcast application; this proof requires reasoning only about individual state transitions. The performance and fault-tolerance properties of the specification are used in an operational proof of the performance and fault-tolerance properties of the application.

1 Introduction

Background

In the development of practical distributed systems, considerable effort is devoted to making distributed applications robust in the face of typical processor and communication failures. Constructing such systems is difficult, however, because of the complexities of the applications and of the fault-prone distributed settings in which they run. To aid in this construction, some computing environments include general-purpose building blocks that provide powerful distributed computation services.

Among the most important examples of building blocks are *group communication services*. Group communication services enable processes located at different nodes of a distributed network to operate collectively as a group; the processes do this by using a group communication service to multicast messages to all members of the group. Different group communication services offer different guarantees about the order and reliability of message delivery. Examples are found in Isis [6], Transis [11], Totem [25], Newtop [13], Relacs [3] and Horus [27].

The basis of a group communication service is a *group membership service*. Each process, at each time, has a unique *view* of the membership of the group. The view includes a list of the processes that are members of the group. Views can change from time to time, and may become different at different processes. Isis introduced the important concept of *virtual synchrony* [6]. This concept has been interpreted in various ways, but an essential requirement is that if a particular message is delivered to several processes, then all have the same view of the membership when the message is delivered. This allows the recipients to take coordinated action based on the message, the membership set and the rules prescribed by the application.

The Isis system was designed for an environment where processors might fail and messages might be lost, but where the network does not partition. That is, it assumes that there are never two disjoint sets of processors, each set communicating successfully among its members. This assumption might be reasonable for some local area networks, but it is not valid in wide area networks. Therefore, the more recent systems mentioned above allow the possibility that concurrent views of the group might be disjoint.

To be most useful to application programmers, system building blocks should come equipped with simple and precise specifications of their guaranteed behavior. These specifications should include not only safety properties, but also performance and fault-tolerance properties. Such specifications would allow application programmers to think carefully about the behavior of systems that use the primitives, without having to understand how the primitives themselves are implemented. Unfortunately, providing appropriate specifications for group communication services is not an easy task. Some of these services are rather complicated, and there is still no agreement about exactly what the guarantees should be. Different specifications arise from different implementations of the same service, because of differences in the safety, performance, or fault-tolerance that is provided. Moreover, the specifications that most accurately describe particular implementations may not be the ones that are easiest for application programmers to use.

The first major work on the development of specifications for fault-tolerant group-oriented membership and communication services appears to be that of Ricciardi [28], and the research area is still active (see, e.g., [26, 7]). In particular, there has been a large amount of work on developing specifications for partitionable group services. Some specifications deal just with membership and views [17, 29] while others also cover message services (ordering and reliability properties) [24, 4, 5, 9, 12, 15, 16]. These specifications are all complicated, many are difficult to understand, and some seem to be ambiguous. It is not clear how to tell whether a specification is sufficient for a given application. It is not even clear how to tell whether a specification is implementable at all; impossibility results such as those in [7] demonstrate that this is a significant issue.

Our contributions

We present a new, simple formal specification for a partitionable view-oriented group communication service. To demonstrate the value of our specification, we use it to construct an ordered-broadcast application, using an algorithm, based on algorithms of Amir, Dolev, Keidar, Melliar-Smith and Moser [18, 1], that reconciles information derived from different views. We prove the correctness and analyze the performance and fault-tolerance of this algorithm. Our specification has a simple implementation, based on the membership algorithm of Cristian and Schmuck [10]. We call our specification *VS*, which stands for *view-synchrony*.[1]

In *VS*, the views are presented to each processor[2] according to a consistent total order, though not every processor need see every view. Each message is associated with a particular view, and all send and receive events for a message occur at processors when they have the associated view. The service provides a total order on the messages associated with each view, and each processor receives a prefix of this total order. There are also some guarantees about stabilization of view information and about successful message delivery, under certain assumptions about the number of failures and about the stabilization of failure behavior.

Our specification *VS* does not describe all the potentially-useful properties of any particular implementation. Rather, it includes only the properties that are needed for the ordered-broadcast application. However, preliminary results suggest that the same specification is also useful for other applications.

The style of our specification is different from those of previous specifications for group communication services, in that we separate safety requirements from performance and fault-tolerance requirements. The safety requirements are formulated in terms of an abstract, global input/output state machine, using precondition-effect notation. This enables assertional reasoning about systems that use this service. The performance and fault-tolerance requirements are expressed as a collection of properties that must hold in executions of the service. Specifically, we include *failure-status input actions* in the specification; we then give properties saying that consensus on the view and timely message delivery are guaranteed in an execution provided that it *stabilizes* to a situation in which the failure status stops changing and corresponds to a consistently partitioned system. This stabilization hypothesis can be seen as an abstract version of the "timed asynchronous model" of Cristian [8]. These performance and fault-tolerance properties are expressed in precise natural language and require operational reasoning.

We consider how our view-synchronous group communication service can be used in the distributed implementation of a sequentially consistent memory. It turns out that the problem can be subdivided into two: the implementation of a *totally ordered broadcast* communication service using a view-synchronous group communication service, and the implementation of sequentially consistent memory using a totally ordered broadcast service. The second of these is easy using known techniques,[3] so we focus in this paper on the first problem. A totally ordered broadcast service delivers messages submitted by its clients, according to a single total ordering of all the messages; this total order must be consistent with the order in which the messages are sent by any particular sender. Each client receives a prefix of the ordering, and there are also some guarantees of successful delivery, under certain assumptions about the stabilization of failure behavior. This service is different from a view-synchronous group communication

[1] This is not the same as the notion of view-synchrony defined in [5].

[2] We consider "processor groups" in the formal material of this paper rather than "process groups". The distinction is unimportant here.

[3] Each processor maintains a replica of the underlying memory. A read operation is performed immediately on the local copy. A requested update is sent to all processors via the totally ordered broadcast service. Each processor (even the one where the request was submitted) applies the update when it is delivered by the totally ordered broadcast service; the submitting processor also determines the return value and returns it to the client. The fact that this provides a sequentially consistent shared memory is at the heart of the "Replicated State Machine" approach to distributed system design. It was first described by Lamport [19], a survey of this approach is given by Schneider in [30] (see also the references therein). An alternative approach is to send all operations (not just updates) through the totally ordered broadcast service; this approach constructs an atomic shared memory.

service in that there is no notion of "view"; the ordering guarantees apply to all the messages, not just those within individual views.

We begin in Section 3 by giving a simple formal specification for a totally ordered broadcast service, which we call *TO*. This specification will be used later as the correctness definition for an algorithm running over a group communication service. It also serves as a simple example to illustrate the style of specification we use throughout the paper: an abstract state machine for safety properties, plus stabilized properties for performance and fault-tolerance.

Then, in Section 4, we present our new specification for a partitionable group communication service, *VS*. In *VS*, there is a crisp notion of a *local view*, that is, each processor, at any time, has a current view and knows the membership of the group in its current view; moreover, any messages sent by any processor in a view are received (if they are received at all) in the same view. The *VS* layer also provides a "safe" indication, once a message has been delivered to all members of the view.

Here are the most important differences between our specification *VS* and other group communication specifications.

1. *VS* does not mention any "transitional views" or "hidden views", such as are found in Extended Virtual Synchrony [24] or the specification of Dolev et al [12]. Each processor always has a well-defined view of the group membership, and all recipients of a message share the view that the sender had when the message was sent.

2. *VS* does not require that a processor learn of all the views of which it is a member.

3. *VS* does not require any relationship among the membership of concurrent views held by different processors. Stronger specifications demand that these views be either disjoint or identical [5], or either disjoint or subsets [4].

4. *VS* does not require consensus on whether a message is delivered. Many other specifications for group communication, including [4, 5, 12, 15, 24], insist on delivery at every processor in the intersection of the current view and a successor view. We allow each member to receive a different subset of the messages associated with the view; however, each member must receive a prefix of a common total order of the messages of that view.

5. The "safe" indication is separate from the message delivery event. In Transis, Totem and Horus [11, 25, 27], delivery can be delayed until the lower layer at each site has the message (though it might not yet have delivered it). Thus in these systems, safe delivery means that every other member is guaranteed to also provide safe delivery or crash. A simple "coordinated attack" argument (as in Chapter 5 of [20]) shows that in a partitionable system, this notion of safe delivery is incompatible with having all recipients in exactly the same view as the sender. In contrast, our service delivers a message before it is safe and later provides a notification once delivery has happened at all other group members.

6. There are no liveness requirements that apply to all executions. Instead, we follow the "timed asynchronous model" of Cristian [8] and make conditional claims for timely delivery only in certain executions where the processors and links behave well.

7. *VS* does not require that every view change have a cause; in contrast, some specifications require that the removal of a member that was in the previous view must be due to a failure of that member, or of a link to it. We allow arbitrary view changes during periods when the underlying network is unstable, however the conditional performance and fault-tolerance property of *VS* shows that once the communication stabilizes to a consistently partitioned system, process views must quickly converge to match that partioning.

The differences represented by points 2, 4 and 6 mean that our *VS* service is not subject to the impossibility results that afflict some group communication specifications [5, 7].

Although *VS* is weaker in several respects than most considered in the literature, we demonstrate that it is strong enough to be useful, by showing, in Section 5, how an interesting and useful algorithm can run on top of it. This algorithm is based on data replication algorithms developed by Amir, Dolev,

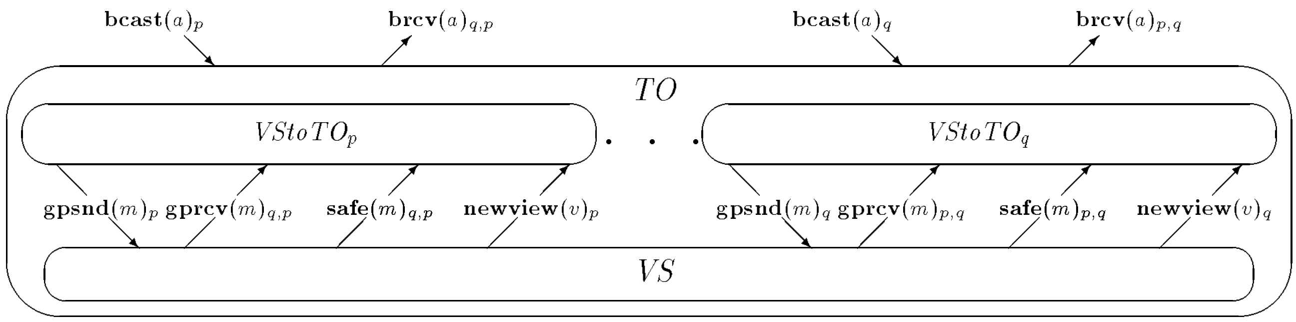

Figure 1: System components and interfaces

Keidar, Melliar-Smith and Moser [18, 1]. These previous algorithms implement a fault-tolerant shared memory by sending modification operations to each replica through a group communication service based on Extended Virtual Synchrony, and carrying out a state-exchange protocol when partition components merge. Our algorithm, which we call *VStoTO*, can be seen as a more abstract form of both previous ones, separated from the specific use for data replication. We present the algorithm using I/O automata [21, 20].

Figure 1 depicts the major components of the system we consider, and their interactions.

Finally, in Sections 6 and 7, we give a proof that the *VStoTO* algorithm, running on top of *VS*, indeed provides the service expressed by the *TO* specification. The safety aspect of this claim uses assertional methods. We give invariants on the global state of a system that consists of the *VStoTO* algorithm and the *VS* state machine. We then give a simulation relationship between the global state of the system, and the *TO* state machine. As usual, proving the invariants and the simulation relationship involves reasoning only about individual state transitions; it does not require operational reasoning, in which one considers a whole execution. The performance and fault-tolerance aspects of the proof involve operational reasoning about timed executions.

2 Mathematical Foundation

If r is a binary relation, then we define $dom(r)$ to be the set (without repetitions) of first elements of the ordered pairs comprising relation r, and $range(r)$ to be the set of second elements. If f is a partial function from A to B and $\langle a, b\rangle \in A \times B$, then $f \oplus \langle a, b\rangle$ is defined to be the partial function that is identical to f except that $f(a) = b$.

If f and g are partial functions, from A to B and from A to C respectively, then the pair $\langle f, g\rangle$ is defined to be the function from A to $B \times C$ such that $\langle f, g\rangle(a) = \langle f(a), g(a)\rangle$.

We write λ for the empty sequence, and $\langle\langle a\rangle\rangle$ for the sequence consisting of the single element a. If s is a sequence, $length(s)$ denotes the length of s. If s is a sequence and $1 \leq i \leq length(s)$ then $s(i)$ denotes the ith element of s. If s and t are sequences and s is finite, then the concatenation of s and t is denoted by $s \cdot t$. We say that sequence s is a *prefix* of sequence t, written as $s \leq t$, provided that there exists s' such that $s \cdot s' = t$. A collection S of sequences is *consistent* provided that for every $s, t \in S$, either $s \leq t$ or $t \leq s$. If S is a consistent collection of sequences, we define $lub(S)$ to be the minimum sequence t such that $s \leq t$ for all $s \in S$.

We often regard a sequence s as a partial function from its index set to its elements; thus, for example, we use the function notation $range(s)$ to denote the set of elements appearing in sequence s. If s is a sequence of elements of X and f is a partial function from X to Y whose domain includes $range(s)$, then $applyall(f, s)$ denotes the sequence t of elements of Y such that $length(t) = length(s)$ and, for $i \leq length(t)$, $t(i) = f(s(i))$.

Our services and algorithms are described using untimed and timed state machine models. Untimed models are used for the safety properties, while timed models are used for the performance and fault-tolerance properties.

The untimed model we use is the I/O automaton model of Lynch and Tuttle [21], also described in

Chapter 8 of [20]. We do not use the "task" construct of the model – the only components we need are a set of states, a designated subset of start states, a signature specifying input, output and internal actions, and a set of (state,action,state) transitions. The timed model we use is that of Lynch and Vaandrager [23], as described in Chapter 23 of [20]. This is similar to the untimed model, but also includes *time passage actions* $\nu(t)$, which indicate the passage of real time t. Time passage actions also have associated state transitions.

An *execution fragment* of an I/O automaton is an alternating sequence of states and actions consistent with the transition relation. An *execution* is an execution fragment that begins with a start state. *Timed execution fragments* and *timed executions* of a timed automaton are defined in the same way. A timed execution fragment of a timed automaton has a "limit time" $ltime \in \mathcal{R}^{\geq 0} \cup \{\infty\}$, which is the sum of all the amounts of time in its time passage actions.

Since our treatment is compositional, we need notions of external behavior for both types of automata. For I/O automata, we use *traces*, which are sequences of actions; for timed automata, we use *timed traces*, each of which is a sequence of actions paired with its time of occurrence, together with a value $ltime \in \mathcal{R}^{\geq 0} \cup \{\infty\}$ indicating the total duration of time over which the events are observed. The external behavior of an I/O automaton is captured by the set of traces generated by its executions, while that of a timed automaton is captured by the set of timed traces generated by its "admissible" timed executions, i.e., those in which $ltime = \infty$.

Execution fragments can be concatenated, as can timed execution fragments, traces and timed traces. I/O automata can be composed, as can timed automata; Chapters 8 and 23 of [20] contain theorems showing that composition respects the external behavior. Invariant assertion and simulation relation methods for these two models are also presented in those chapters.

3 Totally Ordered Broadcast

In this section, we present *TO*, our specification for a totally ordered broadcast communication service. *TO* is a combination of a state machine *TO-machine* and a performance/fault-tolerance property *TO-property*, which is a property of timed traces allowed by a timed version of *TO-machine*.

For the rest of the paper, we fix P to be a totally ordered finite set of processor identifiers (we will often refer to these as *locations*), and A to be a set of data values.

3.1 The State Machine *TO-machine*

The interface between the totally ordered broadcast service and its clients is through input actions of the form $\mathbf{bcast}(a)_p$, representing the submission of data value a by a client at the location of processor p, and output actions of the form $\mathbf{brcv}(a)_{p,q}$, representing the delivery to a client at q of a data value previously sent by a client at p. We call the messages at this interface "data values", to distinguish them from messages at lower-level interfaces.

The state of the specification automaton includes a queue *queue* of data values, each paired with the location at which it originated; the order represented by *queue* is determined by the service implemented by the *TO-machine*. Also, for each location p, there is a queue $pending[p]$ containing the data values originating at p that have not yet been added to *queue*. Finally, for each p there is an integer $next[p]$ giving the index in *queue* of the next data value to be delivered at p. The formal automaton definition is given in Figure 2.

The finite traces of this automaton are exactly the finite prefixes of traces of a totally ordered causal broadcast layer, as defined in [14].

Note that, in any trace of *TO-machine*, there is a natural correspondence between **brcv** events and the **bcast** events that cause them.

TO-machine:

Signature:

Input:
  $\mathbf{bcast}(a)_p$, $a \in A$, $p \in P$
Output:
  $\mathbf{brcv}(a)_{p,q}$, $a \in A$, $p, q \in P$

Internal:
  $\mathbf{to\text{-}order}(a, p)$, $a \in A$, $p \in P$

States:

$queue$, a finite sequence of $A \times P$, initially empty
for each $p \in P$:
  $pending[p]$, a finite sequence of A, initially empty
  $next[p] \in \mathcal{N}^{>0}$, initially 1

Transitions:

$\mathbf{bcast}(a)_p$
Effect:
  append a to $pending[p]$

$\mathbf{to\text{-}order}(a, p)$
Precondition:
  a is head of $pending[p]$
Effect:
  remove head of $pending[p]$
  append $\langle a, p\rangle$ to $queue$

$\mathbf{brcv}(a)_{p,q}$
Precondition:
  $queue(next[q]) = \langle a, p\rangle$
Effect:
  $next[q] \leftarrow next[q] + 1$

Figure 2: *TO-machine*

3.2 The Performance and Fault-Tolerance Property *TO-property*

Consider a signature *TO-fsig* that is the same as that of *TO-machine*, above, with the addition of the following actions:

Input:

for each p:	for each p, q:
$\mathbf{good}_p$	$\mathbf{good}_{p,q}$
$\mathbf{bad}_p$	$\mathbf{bad}_{p,q}$
$\mathbf{ugly}_p$	$\mathbf{ugly}_{p,q}$

If β is any finite sequence of actions of *TO-fsig*, then we define the failure status of any location or pair of locations after β to be either *good*, *bad*, or *ugly*, based on the last action for that location or pair of locations in β. If there is no such action, the default choice is *good*. We extend this definition to related types, e.g., where β is a sequence of timed actions.

The intention (though this has no formal meaning at the level of an abstract specification) is that a *good* process takes steps with no time delay after they become enabled, a *bad* process is stopped, and an *ugly* process operates at nondeterministic speed (or may even stop). Similarly, a *good* channel delivers all messages that are sent while it is good, within a fixed time of sending. A *bad* channel delivers no messages. An *ugly* channel might or might not deliver its messages, and there are no timing restrictions on delivery. But these statements refer to processors, channels and their properties, notions that belong in an implementation model, not in an abstract service specification.

To formulate our performance/fault-tolerance claim, we define *TO-property*(b, d, Q) as a parameterized property of a *timed sequence pair* over external actions of *TO-fsig*, as defined in [23]. This is a pair consisting of a sequence β of timed actions (with non-decreasing times) together with an *ltime*. Here, we only consider cases where $ltime = \infty$. The parameters b and d are nonnegative reals, and the parameter Q is a set of processors.

TO-property(b, d, Q):
Both of the following hold:

1. β with the timing information removed is a trace of *TO-machine*.
2. Suppose that $(\beta, \infty) = (\gamma, l)(\delta, \infty)$ and that all the following hold:

   (a) δ contains no failure status events for locations in Q or for pairs including a location in Q.

(b) All locations in Q and all pairs of locations in Q are *good* after γ.

(c) If $p \in Q$ and $q \notin Q$ then (p, q) is *bad* after γ.

Then (δ, ∞) can be written as $(\delta', l')(\delta'', \infty)$, where

(a) $l' \leq b$.

(b) Every data value sent from a location in Q in β at time t is delivered at all members of Q by time $\max(t, (l + l')) + d$.

(c) Every data value delivered in β to any location in Q at time t is delivered at all members of Q by time $\max(t, (l + l')) + d$.

3.2.1 The Combined Specification TO

We define the specification $TO(b, d, Q)$ to be the pair consisting of *TO-machine* and *TO-property*(b, d, Q).

We say that a timed automaton A *satisfies* the specification $TO(b, d, Q)$ provided that every admissible timed trace of A is in the set (of timed sequence pairs) defined by *TO-property*(b, d, Q).

4 View-Synchronous Group Communication

In this section, we give a formal specification for our view-synchronous group communication service. This specification is again based on a combination of a state machine, *VS-machine*, and a performance/fault-tolerance property, *VS-property*.

For the rest of the paper, we fix M to be a message alphabet, and $\langle G, <_G, g_0 \rangle$ to be a totally ordered set of view identifiers with an initial view identifier. We define $views = G \times \mathcal{P}(P)$, the set of pairs consisting of a view identifier together with a set of locations; an element of the set *views* is called a *view*. If v is a view, we write $v.id$ and $v.set$ to denote the view identifier and set components of v, respectively.

4.1 The State Machine *VS-machine*

The external actions of *VS-machine* include actions of the form $\mathbf{gpsnd}(m)_p$, representing the client at p sending a message m, and actions of the form $\mathbf{gprcv}(m)_{p,q}$, representing the delivery to q of the message m sent by p. Outputs $\mathbf{safe}(m)_{p,q}$ are also provided at q to report that the earlier message m from p has been delivered to all locations in the current view as known by q.

VS-machine informs its clients of group status changes through $\mathbf{newview}(\langle g, S \rangle)_p$ actions (with $p \in S$), which tells p that the view identifier g is associated with membership set S and that, until another **newview** occurs, the following messages will be in this view. After any finite execution, we define the current view at p to be the argument v in the last $\mathbf{newview}_p$ event, if any, otherwise it is the pair consisting of the distinguished initial view identifier g_0 and the universe P of processor locations.

The code is given in Figure 3. The state of the automaton is similar to that of *TO-machine*, except that there are multiple queues, one per view identifier, and similarly for each view identifier there is a separate indicator for the next index to be delivered to a given location. Also, the service keeps track of all the views that have ever been defined, and of the current view at each location.

The actions for creating a view and for informing a processor of a new view are straightforward (recall that the signature ensures that only members, but not necessarily all members, receive notification of a new view). Within each view, messages are handled as in *TO-machine*: first kept pending, then placed into a total order in the appropriate queue, and finally passed to the environment. Thus, *VS-machine* ensures that each $\mathbf{gprcv}_{p,q}$ and each $\mathbf{safe}_{p,q}$ event occurs at q when q's view is the same as p's view when the corresponding **gpsnd** event occurs. The specification given in Figure 3 (unlike the particular *VStoTO* algorithm presented later) does not have any notion of "primary" view: it does not treat a message associated with a majority view differently from one in a minority view.

Note that *VS-machine* does not include any restrictions on when a new view might be formed. However, our performance and fault-tolerance property *VS-prop*, described below, does express such restrictions – it implies that "capricious" view changes must stop shortly after the behavior of the underlying physical system stabilizes. In any trace of *VS-machine*, there is a natural correspondence

VS-machine:

Signature:

Input:
  $\mathbf{gpsnd}(m)_p$, $m \in M$, $p \in P$
Output:
  $\mathbf{gprcv}(m)_{p,q}$ hidden g, $m \in M$, $p \in P$, $q \in P$, $g \in G$
  $\mathbf{safe}(m)_{p,q}$ hidden v, $m \in M$, $p \in P$, $q \in P$, $v \in views$
  $\mathbf{newview}(v)_p$, $v \in views$, $p \in P$, $p \in v.set$

Internal:
  $\mathbf{createview}(v)$, $v \in views$
  $\mathbf{vs\text{-}order}(m, p, g)$, $m \in M$, $p \in P$, $g \in G$

States:

$created \subseteq views$, initially $\{\langle g_0, P\rangle\}$
for each $p \in P$:
  $current\text{-}viewid[p] \in G$, initially g_0
for each $g \in G$:
  $queue[g]$, a finite sequence of $M \times P$, initially empty

for each $p \in P$, $g \in G$:
  $pending[p, g]$, a finite sequence of M, initially empty
  $next[p, g] \in \mathcal{N}^{>0}$, initially 1
  $next\text{-}safe[p, g] \in \mathcal{N}^{>0}$, initially 1

Transitions:

$\mathbf{createview}(v)$
Precondition:
  $v.id > \max\{g : \exists S : \langle g, S\rangle \in created\}$
Effect:
  $created \leftarrow created \cup \{v\}$

$\mathbf{newview}(v)_p$
Precondition:
  $v \in created$
  $v.id > current\text{-}viewid[p]$
Effect:
  $current\text{-}viewid[p] \leftarrow v.id$

$\mathbf{gpsnd}(m)_p$
Effect:
  append m to $pending[p, current\text{-}viewid[p]]$

$\mathbf{vs\text{-}order}(m, p, g)$
Precondition:
  m is head of $pending[p, g]$
Effect:
  remove head of $pending[p, g]$
  append $\langle m, p\rangle$ to $queue[g]$

$\mathbf{gprcv}(m)_{p,q}$, hidden g
Precondition:
  $g = current\text{-}viewid[q]$
  $queue[g](next[q, g]) = \langle m, p\rangle$
Effect:
  $next[q, g] \leftarrow next[q, g] + 1$

$\mathbf{safe}(m)_{p,q}$, hidden g, S
Precondition:
  $g = current\text{-}viewid[q]$
  $\langle g, S\rangle \in created$
  $queue[g](next\text{-}safe[q, g]) = \langle m, p\rangle$
  for all $r \in S$:
    $next[r, g] > next\text{-}safe[q, g]$
Effect:
  $next\text{-}safe[q, g] \leftarrow next\text{-}safe[q, g] + 1$

Figure 3: *VS-machine*

between **gprcv** events and the **gpsnd** events that cause them, and between **safe** events and the **gpsnd** events that cause them.

For later use we give some facts about *VS-machine*. These are expressed using a derived variable. Derived variables:

$created\text{-}viewids = \{g \in G | \exists S : \langle g, S \rangle \in created\}$

Lemma 4.1 *The following are true in all reachable states of VS-machine: For any $p \in P$, $S \subseteq P$, $m \in M$, $g \in G$:*

1. *If $g \in created\text{-}viewids$ then there is a unique S such that $\langle g, S\rangle \in created$.*
2. *$current\text{-}viewid[p] \in created\text{-}viewids$.*
3. *If $\langle current\text{-}viewid[p], S\rangle \in created$ then $p \in S$.*
4. *If $pending[p, g] \neq \lambda$ then $g \in created\text{-}viewids$.*
5. *If $pending[p, g] \neq \lambda$ then $g \leq current\text{-}viewid[p]$.*
6. *If $queue[g] \neq \lambda$ then $g \in created\text{-}viewids$.*
7. *If $\langle m, p\rangle$ is in $queue[g]$ then $g \leq current\text{-}viewid[p]$.*
8. *$next[p, g] \leq length(queue[g]) + 1$.*
9. *$next\text{-}safe[p, g] \leq length(queue[g]) + 1$.*
10. *$next\text{-}safe[p, q] \leq next[p, g]$.*
11. *If $\langle g, S\rangle \in created$ and $next[p, g] \neq 1$ then $p \in S$.*
12. *If $\langle g, S\rangle \in created$ and $next\text{-}safe[p, g] \neq 1$ then $p \in S$.*

Proof. All are straightforward by induction. □

Lemma 4.2 *If β is any trace of VS-machine, then there exists a unique total function cause mapping* **gprcv** *and* **safe** *events in β to* **gpsnd** *events in β, such that:*

1. *(Message integrity) For each π, cause(π) precedes π, has the same value argument, and has a processor subscript equal to the first ("source") subscript of π. Moreover, the current view at the location of π when π occurs is the same as the current view at the location of cause(π) when cause(π) occurs.*
2. *(No duplication) For each q, cause is one-to-one for* **gprcv** *events with second ("destination") subscript q. Similarly, for each q, cause is one-to-one for* **safe** *events with second ("destination") subscript q.*
3. *(No reordering) For each p and q, and within each view: cause is monotone increasing for* $\mathbf{gprcv}_{p,q}$ *events, and cause is monotone increasing for* $\mathbf{safe}_{p,q}$ *events.*
4. *(No losses) For each p and q, and within each view: The range of cause for* $\mathbf{gprcv}_{p,q}$ *events is a prefix of the subsequence of* $\mathbf{gpsnd}_p$ *events, and the range of cause for* $\mathbf{safe}_{p,q}$ *events is a prefix of the subsequence of* $\mathbf{gpsnd}_p$ *events.*

Proof. Such a *cause* mapping can be constructed from any execution α that gives rise to trace β, by assigning uids to **gpsnd** events and carrying them along in the *pending* and *queue* components.

Uniqueness is immediate, since the properties require that the i-th $\mathbf{gprcv}_{p,q}$ event within a particular view g must be mapped to the i-th $\mathbf{gpsnd}_p$ event within the same view g, and similarly for $\mathbf{safe}_{p,q}$ events. □

Lemma 4.2 allows us to implicitly associate a particular **gpsnd** event with each **gprcv** event and each **safe** event, in any trace of *VS-machine*.

As an alternative possibility for specifying view-synchronous group communication, we might weaken the createview precondition so that it only enforces unique ids, and does not enforce in-order creation:

createview(v)
Precondition:
  for all $w \in created$,
    $v.id \neq w.id$
Effect:
  $created \leftarrow created \cup \{v\}$

We call this alternative specification *WeakVS-machine*. We do not prove it here, but in fact *Weak-VS-machine* and *VS-machine* are equivalent specifications: they allow exactly the same finite traces. Thus, the safety property for the *VStoTO-system* remains valid when using *WeakVS-machine* in place of *VS-machine*.

4.2 The Performance and Fault-Tolerance Property *VS-property*

Consider a signature *VS-fsig* that is the same as that of *VS-machine*, above, with the addition of the failure status actions (as before). We define *VS-property* as a parameterized property of a *timed sequence pair* (β, ∞) over external actions of *VS-fsig*. The parameterized property is defined as follows. Parameters b and d are nonnegative reals, and Q is a set of processors.

VS-property(b, d, Q):
Both of the following hold:

1. β with the timing information removed is a trace of *VS-machine*.
2. Suppose that $(\beta, \infty) = (\gamma, l)(\delta, \infty)$. Suppose that all the following hold:
   (a) δ contains no failure status events for locations in Q or for pairs including a location in Q.
   (b) All locations in Q and all pairs of locations in Q are *good* after γ.
   (c) If $p \in Q$ and $q \notin Q$ then (p, q) is *bad* after γ.

   Then (δ, ∞) can be written as $(\delta', l')(\delta'', \infty)$, where
   (a) $l' \leq b$
   (b) No **newview** events occur in δ'' at locations in Q.
   (c) The latest views at all locations in Q after $\gamma\delta'$ are the same, say $\langle g, S\rangle$, where $S = Q$.
   (d) Every message sent from a location in Q in β while in view $\langle g, S\rangle$ at time t has corresponding *safe* events at all members of Q by time $\max(t, (l + l')) + d$.

4.2.1 The Combined Specification *VS*

We define the specification $VS(b, d, Q)$ to be the pair consisting of *VS-machine* and *VS-property*(b, d, Q).

We say that a timed automaton A *satisfies* the specification $VS(b, d, Q)$ provided that every admissible timed trace of A is in the set defined by *VS-property*(b, d, Q).

5 The algorithm *VStoTO*

Now we describe the *VStoTO* algorithm, which uses *VS* to implement *TO*. As depicted in Figure 1, the algorithm consists of an automaton $VStoTO_p$ for each $p \in P$. Code for $VStoTO_p$ appears in Figure 5, and some auxiliary definitions needed in the code appear in Figure 4.

For the rest of the paper, we fix a set $\mathcal{Q}$ of *quorums*, each of which is a subset of P. We assume that every pair Q, Q' in $\mathcal{Q}$ satisfy $Q \cap Q' \neq \emptyset$.

The activities of the algorithm consist of *normal* activity and *recovery* activity. Normal activity occurs while group views are stable. Recovery activity begins when a new view is presented by *VS*, and continues while the members exchange and combine information from their previous histories in order to establish a consistent basis for subsequent normal activity.

In the normal case, each value received by $VStoTO_p$ from the client is assigned a system-wide unique *label* consisting of the viewid at p when the value arrives, a sequence number, and the processor id p. The variable *current* keeps track of the current view, and the variable *nextseqno* is used to generate the sequence numbers. Labels are ordered lexicographically. $VStoTO_p$ stores the $\langle$ label,value $\rangle$ pair in

Types:
$L = G \times \mathcal{N}^{>0} \times P$, with selectors *id*, *seqno*, *origin*
$summaries = \mathcal{P}(L \times A) \times (L^*) \times \mathcal{N}^{>0} \times G$, with selectors *con*, *ord*, *next*, and *high*
Operations on types:
For $x \in summaries$,
$x.confirm$ is the prefix of $x.ord$ such that $length(x.confirm) = min(x.next - 1, length(x.ord))$
For Y a partial function from processor ids to summaries,
$knowncontent(Y) = \cup_{q \in dom(Y)} Y(q).con$
$maxprimary(Y) = \max_{q \in dom(Y)}\{Y(q).high\}$
$reps(Y) = \{q \in dom(Y) : Y(q).high = maxprimary\}$
$chosenrep(Y)$ is some element in $reps(Y)$
$shortorder(Y) = Y(chosenrep(Y)).ord$
$fullorder(Y)$ is $shortorder(Y)$ followed by the remaining elements of $dom(knowncontent(Y))$, in label order
$maxnextconfirm(Y) = \max_{q \in dom(Y)} Y(q).next$

Figure 4: Definitions used in *VStoTO* automaton

a relation *content*. It sends the pair to the other members of the current view, using *VS*, and these other processors also add the pair to their own *content* relations. An invariant shows that each *content* relation is actually a partial function from labels to values, and that a given label is associated with the same data value everywhere.

The algorithm distinguishes *primary* views, whose membership includes a quorum of processors, from *non-primary* views. When $VStoTO_p$ receives a ⟨ label,value ⟩ pair while it is in a primary view, it places the label at the end of its sequence *order*. In combination with *content*, *order* describes a total order of submitted data values; this represents a tentative version of the system-wide total ordering of data values that the *TO* service is supposed to provide. The consistent order of message delivery within each view (guaranteed by *VS*) ensures that *order* is consistent among members of a particular view, but it need not always be consistent among processors in different views. When $VStoTO_p$ receives a ⟨ label,value ⟩ pair while it is in a non-primary view, it does not process the pair (except for recording it in *content*).

$VStoTO_p$ remembers which data values have been reported as safely delivered to all members of the current view, using a set *safe-labels* of labels. When a label is in *safe-labels*, it is a candidate for becoming "confirmed" for release to the client. Labels in the *order* sequence become confirmed in the same order in which they appear in *order*. The variable *nextconfirm* is used to keep track of the prefix of the current *order* sequence that is confirmed. $VStoTO_p$ can release data values associated with confirmed labels to the client, in the order described by *order*. The variable *nextreport* is used to keep track of which values have been released to the client.

Recovery activity begins when *VS* performs a **newview** event. This activity involves exchanging and combining information to integrate the knowledge of different members of the new view. The recovery process consists of two, possibly overlapping phases. In the first phase of recovery, each member of a new view uses *VS* to send a *state-exchange* message containing a summary of that processor's state, including the values of its *content*, *order* and *nextconfirm* variables. In order to use this state information, each processor must determine which member has the most up-to-date information. For this purpose, another variable *highprimary* is used to record the highest view identifier of a primary view in which an *order* was calculated that has affected the processor's own *order* sequence. (This effect can be through the processor's own earlier participation in that primary view, or through indirect information in previous state exchange messages.) The value of the *highprimary* variable is also included in the summary sent in the state-exchange message.

During this first phase of recovery, $VStoTO_p$ records the summary information received from the other members of the new view, in *gotstate*, which is a partial function from processor ids to summaries. Once $VStoTO_p$ has collected all members' summaries, it processes the information in one atomic step;

at this point, it is said to *establish* the new view. The processor processes state information by first defining its confirmed labels to be longest prefix of confirmed labels known in any of the summaries. Then it determines the *representatives*, which are the members whose summaries include the greatest *highprimary* value. Then the information is processed in different ways, depending on whether or not the new view is primary.

If the new view is not primary, the processor adopts as its new *order* the *order* sent by a particular "chosen" representative processor. In this case, *highprimary* is set equal to the greatest *highprimary* in any of the summaries, i.e., the *highprimary* of the chosen representative. On the other hand, if the view is primary, the processor adopts as its new *order* the *order* computed as above for non-primary views, extended with all other known labels appearing in any of the summaries in *gotstate*, arranged in label order. In this case, *highprimary* is set equal to the new viewid.

Extracting the various pieces of information described above from *gotstate* requires some auxiliary functions, which are defined in Figure 4. Namely, let Y be a value of the type recorded in the *gotstate* component. Then $knowncontent(Y)$ contains all the (label, value) pairs in the summaries recorded in Y. Also, $maxprimary(Y)$ is the greatest view identifier of an established primary appearing in any of the summaries, $reps(Y)$ denotes the set of members that know of this view, and $chosenrep(Y)$ is some consistently-chosen element of this set. (Any method can be used to select the particular representative, as long as all processors select the same one from identical information; for example, they could choose the representative with the highest processor id, or the one with the shortest or longest *order* sequence.) Now *shortorder* is the *order* of the chosen representative; this is the *order* adopted in a non-primary view, as described above. And *fullorder* consists of $shortorder(Y)$ followed by the remaining elements of $knowncontent(Y)$, in label order; this is the *order* adopted in a primary view. We also define $maxnextconfirm(Y)$ to be the highest among the reported *nextconfirm* values in the exchanged state.

At this point, the first phase of recovery is completed, and normal processing of new client messages is allowed to resume. However, for a primary view, there is a second phase of recovery, which involves collecting the *VS* safe indications for the state-exchange messages. $VStoTO_p$ remembers these indications in a variable *safe-exch*. This phase may overlap with the summary collection phase. Once the state exchange is safe, all labels used in the exchange are marked as safe, and all associated messages are confirmed just as they would be in normal processing. For a non-primary view, there is no second phase of recovery, i.e., the safe indications are ignored.

The state of $VStoTO_p$ also records the *status* of processing, which may be *normal* (anywhere other than in the first phase of recovery), *send* (in the first phase of recovery, after the new view announcement but before sending the state-exchange message), or *collect* (in the first phase of recovery, waiting for some state-exchange messages).

6 Correctness - Safety Argument

Define *VStoTO-system* to be the composition of *VS-machine* and $VStoTO_p$ for all $p \in P$, with the actions used for communication between the two layers (that is, the **gpsnd**, **gprcv**, **safe** and **newview** actions) hidden. In a state of the composition, we refer to the separate state variables by giving a subscript p indicating a variable that is part of the state of $VStoTO_p$.

The proof is based on a forward simulation relation [22] from *VStoTO-system* to *TO-machine*, established with the help of a series of invariant assertions for *VStoTO-system*. We add some derived variables to the state of *VStoTO-system*, for use in defining the simulation relation and in stating and proving the invariants:

We write $allstate[p, g]$ to denote a set of summaries, defined so that $x \in allstate[p, g]$ if and only if at least one of the following hold:

1. $current.id_p = g$ and $x = \langle content_p, order_p, nextconfirm_p, highprimary_p \rangle$.
2. $x \in pending[p, g]$.
3. $\langle x, p \rangle \in queue[g]$.
4. For some q, $current.id_q = g$ and $x = gotstate(p)_q$.

$VStoTO_p$:

Signature:

Input:
- $\mathbf{bcast}(a)_p$, $a \in A$
- $\mathbf{gprcv}(m)_{q,p}$, $q \in P$, $m \in (L \times A) \cup summaries$
- $\mathbf{safe}(m)_{q,p}$, $q \in P$, $m \in (L \times A) \cup summaries$
- $\mathbf{newview}(v)_p$, $v \in views$

Output:
- $\mathbf{gpsnd}(m)_p$, $m \in (L \times A) \cup summaries$
- $\mathbf{brcv}(a)_{q,p}$, $a \in A$, $q \in P$

Internal:
- $\mathbf{confirm}_p$

States:

$current \in views$, initially $\langle g_0, P\rangle$
$status \in \{normal, send, collect\}$, initially $normal$
$content \subseteq L \times A$, initially $\emptyset$
$nextseqno \in \mathcal{N}^{>0}$, initially 1
$buffer$, a finite sequence of elements of L, initially λ
$safe\text{-}labels \subseteq L$, initially $\emptyset$
$order$, a finite sequence of L, initially λ
$nextconfirm \in \mathcal{N}^{>0}$, initially 1

$nextreport \in \mathcal{N}^{>0}$, initially 1
$highprimary \in G$, initially g_0
$gotstate$, a partial function from P to $summaries$, initially $\emptyset$,
$safe\text{-}exch \subseteq P$, initially $\emptyset$,

Derived variables:

$primary$, a Boolean, defined to be the condition that $current.set$ contains some quorum.

Transitions:

$\mathbf{bcast}(a)_p$
Effect:
  $content \leftarrow content \cup \{\langle\langle current.id, nextseqno, p\rangle, a\rangle\}$
  append $\langle current.id, nextseqno, p\rangle$ to $buffer$
  $nextseqno \leftarrow nextseqno + 1$

$\mathbf{gpsnd}(\langle l, a\rangle)_p$
Precondition:
  $status = normal$
  l is head of $buffer$
  $\langle l, a\rangle \in content$
Effect:
  delete head of $buffer$

$\mathbf{gprcv}(\langle l, a\rangle)_{q,p}$
Effect:
  $content \leftarrow content \cup \{\langle l, a\rangle\}$
  if $primary$ then
    $order \leftarrow order \cdot \langle\langle l\rangle\rangle$

$\mathbf{safe}(\langle l, a\rangle)_{q,p}$
Effect:
  if $primary$ then
    $safe\text{-}labels \leftarrow safe\text{-}labels \cup \{l\}$

$\mathbf{confirm}_p$
Precondition:
  $primary$
  $order(nextconfirm) \in safe\text{-}labels$
Effect:
  $nextconfirm \leftarrow nextconfirm + 1$

$\mathbf{brcv}(a)_{q,p}$
Precondition:
  $nextreport < nextconfirm$
  $\langle order(nextreport), a\rangle \in content$
  $q = order(nextreport).origin$
Effect:
  $nextreport \leftarrow nextreport + 1$

$\mathbf{newview}(v)_p$
Effect:
  $current \leftarrow v$
  $nextseqno \leftarrow 1$
  $buffer \leftarrow \lambda$
  $gotstate \leftarrow \emptyset$
  $safe\text{-}exch \leftarrow \emptyset$
  $safe\text{-}labels \leftarrow \emptyset$
  $status \leftarrow send$

$\mathbf{gpsnd}(x)_p$
Precondition:
  $status = send$
  $x = \langle content, order, nextconfirm, highprimary\rangle$
Effect:
  $status \leftarrow collect$

$\mathbf{gprcv}(x)_{q,p}$
Effect:
  $content \leftarrow content \cup x.con$
  $gotstate \leftarrow gotstate \oplus \langle q, x\rangle$
  if $(dom(gotstate) = current.set) \wedge (status = collect)$
  then
    $nextconfirm \leftarrow maxnextconfirm(gotstate)$
    if $primary$ then
      $order \leftarrow fullorder(gotstate)$
      $highprimary \leftarrow current.id$
    else
      $order \leftarrow shortorder(gotstate)$
      $highprimary \leftarrow maxprimary(gotstate)$
    $status \leftarrow normal$

$\mathbf{safe}(x)_{q,p}$
Effect:
  $safe\text{-}exch \leftarrow safe\text{-}exch \cup \{q\}$
  if $safe\text{-}exch = current.set$ and $primary$ then
    $safe\text{-}labels \leftarrow safe\text{-}labels \cup range(fullorder(gotstate))$

Figure 5: $VStoTO_p$

Thus, $allstate[p, g]$ consists of all the summary information that is in the state of p if p's current view is g, plus all the summary information that has been sent out by p in state exchange messages in view g and is now remembered elsewhere among the state components of *VStoTO-system*. Notice that $allstate[p, g]$ consists only of summaries: an ordinary message $\langle l, a\rangle$ is never an element of $allstate[p, g]$. We write $allstate[g]$ to denote $\bigcup_{p\in P} allstate[p, g]$, and $allstate$ to denote $\bigcup_{g\in G} allstate[g]$.

We write $allcontent$ for $\bigcup_{x\in allstate} x.con \cup \{\langle l, a\rangle : \exists g, p : \langle\langle l, a\rangle, p\rangle \in range(queue[g]) \vee \langle l, a\rangle \in range(pending[p, g])\}$. This represents all the information available anywhere that links a label with a corresponding data value.

The invariants also require the addition of some history variables to the state of *VStoTO-system*: For every $g \in G$, $established[g]$ is defined to be a Boolean, initially $true$ if $g = g_0$, otherwise $false$; this variable is maintained by placing the statement $established[current.id] \leftarrow true$ in the effects part of $\mathbf{gprcv}(x)_{q,p}$, just after the assignment $status \leftarrow normal$ (and within the scope of the outer if statement).

For every $p \in P$, $g \in G$, $buildorder[p, g]$ is defined to be a sequence of labels, initially empty; this variable is maintained by following every statement of processor p that assigns to $order$ with another statement $buildorder[p, current.id_p] \leftarrow order$. It follows that if p establishes a view with id g, and later leaves view g for a view with a higher viewid, then forever afterwards, $buildorder[p, g]$ remembers the value of $order_p$ at the point where p left view g.

6.1 Invariants

We first prove a long series of invariants, establishing simple relationships among the state variables, and other properties of the reachable states. As usual, each invariant is proved using induction on the length of an execution, assuming previous invariants.

The first invariant asserts consistency between certain variables of the processes and of *VS*.

Lemma 6.1 *The following are true in all reachable states of VStoTO-system.*
For any $p \in P$:

1. $current.id_p = current\text{-}viewid[p]$.
2. $current_p \in created$.

Proof. Easy induction. □

The next invariant characterizes the labels that occur in various state components.

Lemma 6.2 *The following are true in all reachable states of VStoTO-system.*

1. *If $\langle g', *, p'\rangle$ is in $buffer_p$ then $p = p'$ and $g' = current.id_p$.*
2. *If $\langle\langle g', *, p'\rangle, *\rangle$ is in $pending[p, g]$ then $p = p'$ and $g = g'$.*
3. *If $\langle\langle\langle g', *, p'\rangle, *\rangle, p\rangle$ is in $queue[g]$ then $g = g'$ and $p = p'$.*

Proof. Each part is an immediate induction on the execution, using the previous part. □

The next two invariants justify the way the definition of the simulation relation uses $allcontent$ as a function from labels to data values.

Lemma 6.3 *The following is true in all reachable states of VStoTO-system.*
If $l \in domain(allcontent)$ and $l.origin = p$
then $l < \langle current.id_p, nextseqno_p, p\rangle$.

Proof. The only change is in $\mathbf{bcast}_p$ and the code shows that the new label is less than the new $(current.id_p, nextseqno_p, p)$ triple. □

Lemma 6.4 *The following is true in all reachable states of VStoTO-system.*
allcontent is a function.

Proof. The only change is in **bcast** and Lemma 6.3 shows the new entry is for a new label. □

Lemma 6.5 *The following is true in all reachable states of VStoTO-system.*
If l is in $buffer_p$ then $\langle l, a\rangle$ is in $content_p$ for some a.

Proof. Immediate induction. The only relevant actions are $\mathbf{bcast}_p$ (which adds a new label to *buffer* and to the domain of *content*) and $\mathbf{newview}_p$ (which empties *buffer*). □

The next invariants describe situations when certain information is guaranteed not to appear in the state.

Lemma 6.6 *The following are true in all reachable states of VStoTO-system.*
If $current.id_p < g$ then

1. *$pending[p, g] = \lambda$.*
2. *There is no message of the form $\langle *, p\rangle$ in $queue[g]$.*
3. *If $current.id_q = g$ then there is no $\langle p, *\rangle$ in $gotstate_q$.*
4. *$allstate[p, g] = \emptyset$.*
5. *There is no pair of the form $\langle\langle g, *, p\rangle, *\rangle$ in $x.con$, for any summary $x \in allstate$.*
6. *There is no pair of the form $\langle\langle g, *, p\rangle, *\rangle$ in $content_q$, for any q.*

Proof. Part 1 is a simple induction; part 2 is an induction using part 1 in the **gpsnd**; part 3 is induction using part 2 in the **gprcv**. Part 4 follows from parts 1, 2 and 3. Part 5 is direct from Lemma 6.3; part 6 follows directly from part 5. □

Lemma 6.7 *The following are true in all reachable states of VStoTO-system.*
If $status_p = send$ and $current.id_p = g$ then

1. *$pending[p, g] = \lambda$.*
2. *There is no element of the form $\langle *, p\rangle$ in $queue[g]$.*
3. *If $current.id_q = g$ then there is no $\langle p, *\rangle$ in $gotstate_q$.*
4. *There is no pair of the form $\langle\langle g, *, p\rangle, *\rangle$ in $x.con$, for any summary $x \in allstate$ other than the summary whose components are those from the local state of p.*
5. *There is no pair of the form $\langle\langle g, *, p\rangle, *\rangle$ in $content_q$, for any $q \neq p$.*

Proof. Part 1 is induction using lemma 6.6 for the case $\mathbf{newview}_p$; part 2 is an induction using lemma 6.6 for the case $\mathbf{newview}_p$, and part 1 in the **gpsnd**; part 3 is induction using Lemma 6.6 for $\mathbf{newview}_p$ and and also part 2 in the **gprcv**. Part 4 is induction (where the case of $\mathbf{gpsnd}_p$ is ruled out by the hypothesis on $status_p$); part 5 follows directly from part 4. □

Lemma 6.8 *The following are true in all reachable states of VStoTO-system.*
For any $p \in P$, if $status_p = collect$ and $current.id_p = g$ then the following holds.
If $x \in allstate[p, g]$ then

1. *$x.con \subseteq content_p$.*
2. *$x.ord = order_p$*
3. *$x.next = nextconfirm_p$*
4. *$x.high = highprimary_p$.*

Proof. For parts 1 and 4, when p first enters reaches the *collect* status in view g, the only summary in $allstate[p, g]$ is that whose components are the state of p (we appeal here to Lemma 6.7, since the $status_p = send$ immediately before it becomes *collect*). Thereafter, $content_p$ changes only by union with more pairs, and no action changes $highprimary_p$ without also changing $status_p$ so it is no longer *collect*.

The other parts need more sophisticated proof, as they depend on the property that no ordinary message is received at p until after all members' state exhange messages. As the rest of the paper relies only on part 4, we omit the details. □

Now some simple facts about "established".

Lemma 6.9 *The following are true in all reachable states of VStoTO-system.*
For any $p \in P$, $g \in G$:

1. *If $established[g]_p$ then $current.id_p \geq g$.*
2. *$established[current.id_p]_p$ if and only if $status_p = normal$.*

Proof. Straightforward induction. Depends on views coming in in increasing order. □

Here are some upper bounds on highprimaries.

Lemma 6.10 *The following are true in all reachable states of VStoTO-system.*
For any $p, q \in P$, $x \in summaries$, $g \in G$:

1. *If $established[current.id_p]_p$ and $primary_p$ then $highprimary_p = current.id_p$.*
2. *If $established[current.id_p]_p$ and $not(primary_p)$ then $highprimary_p < current.id_p$.*
3. *If $established[current.id_p]_p = false$ then $highprimary_p < current.id_p$.*
4. *If $\langle q, x\rangle \in gotstate_p$ then $x.high < current.id_p$.*
5. *If $\langle x, q\rangle$ is in $queue[g]$ then $x.high < g$.*
6. *If x is in $pending[q, g]$ then $x.high < g$.*

Proof. Prove all these parts together using induction.

1. When state exchange is completed for a primary view (which establishes $current_p$), the equality is set explicitly.

Supposed $established[current.id_p]$ in both pre- and post-states; the events that could falsify the RHS are newview and a gprcv that sets highprimary. But a $newview(v)_p$ has $s'.status_p = send$, so Lemma 6.9 implies that $s'.established[s'.current.id_p] = false$, a contradiction.

A **gprcv** that sets $highprimary_p$ has $s.status_p = collect$. Then Lemma 6.9 implies that $s.established[s.current.id_p] = false$, a contradiction.

2. Consider a **gprcv** that completes state exchange, and so establishes a non-primary viewid g. The code sets $highprimary_p$ to be the largest high component of the summaries among the prestate's *gotstate* and the final state exchange message. By part 4 and 5, all of these are less than g, hence the largest is also less than g.

3. Consider $newview(v)_p$. LHS becomes true. Claim RHS also true after the step. Parts 1, 2 and 3 together in the pre-state s imply that $s.highprimary_p \leq s.current.id_p$. Since $s'.current.id_p > s.current.id_p$, this means that $s'.highprimary_p < s'.current.id_p$, as needed.

4. Depends on 5. A key fact is that a message only gets delivered if its view is the same as the current view of p.

5. Depends on 6.

6. Depends on 3. Uses the fact that when a process sends, its status is *send*. □

Lemma 6.11 *The following are true in all reachable states of VStoTO-system.*

1. *If $x \in allstate[p, g]$ then $x.high \leq g$.*

2. *If* $x \in allstate[p, g]$ *then* $x.high \leq current.id_p$.

Proof. Part 1 is an easy induction. Part 2 follows using part 4 of Lemma 6.6. □

The next two lemmas assert lower bounds on highprimaries.

Lemma 6.12 *The following is true in all reachable states of VStoTO-system. For any* $p \in P$, $v \in created$, *such that* $established[v.id]_p$, $v.set$ *contains a quorum, and* $current.id_p > v.id$ *then* $highprimary_p \geq v.id$.

Proof. Let $g = v.id$. First consider actions that could make the hypothesis true (we denote the state before the action as s, and that afterwards as s').

1. When $established[g]_p$ becomes $true$, $current.id_p = g$, so the hypothesis is false.

2. Suppose a **newview** step converts the hypothesis from $false$ to $true$. Then $s.established[g]_p = s'.established[g]_p = true$, $s.current.id_p \leq g$ and $s'.current.id_p > g$. Lemma 6.9 implies that $s.current.id_p \geq g$, so it must be that $s.current.id_p = g$. Then Lemma 6.10, part 1 implies that $s.highprimary_p = g$. Therefore, $s'.highprimary_p = g$, as needed.

Now consider steps for which the hypothesis is true both before and after the step, but that make the conclusion false.

3. **gprcv** for a summary, if $domain(s'.gotstate_p) = s.current.set_p$ and $s.status_p = collect$, sets $highprimary_p$.

There are two cases. If $s.primary_p = true$ then since $s'.current.id_p > g$ and $s'.current.id_p = s'.highprimary_p$, we have that $s'.highprimary_p > g$, which suffices.

On the other hand, suppose that $s.primary_p = false$. Since the hypothesis is true before the step, the inductive hypothesis implies that $s.highprimary_p \geq g$. It suffices to show that $s'.highprimary_p \geq s.highprimary_p$.

The step ensures that $s'.highprimary_p = maxprimary(s'.gotstate_p)$. Since $p \in current.set_p$, $s'.gotstate_p$ must include some (p, x). Then part 4 of Lemma 6.8 implies that $x.high = s.highprimary_p$. Now, $maxprimary(s'.gotstate_p) \geq x.high$, so $s'.highprimary \geq s.highprimary$. This suffices. □

Lemma 6.13 *The following are true in all reachable states of VStoTO-system. For any* $p \in P$, *for any summary* x, *and for all* $v, w \in created$:

1. *If* $established[v.id]_p$, $v.set$ *contains a quorum,* $w.id > v.id$, *and* $x \in allstate[p, w.id]$, *then* $x.high \geq v.id$.

Proof. Let g denote $v.id$, and g' denote $w.id$. We need only consider actions that could make the hypothesis true, since the conclusion is unchanged in all transitions.

1. When $established[g]_p$ becomes $true$, $current.id_p = g$. Then part 4 of Lemma 6.6 implies that $allstate[p, g'] = \emptyset$, which makes the conclusion vacuously true.

2. When x first gets into $allstate[p, g']$, this happens by putting it into the state of p when $current.id_p = g' > g$. Then Lemma 6.11 implies that $highprimary_p \geq g$. □

Lemma 6.14 *The following are true in all reachable states of VStoTO-system.*

1. *If* $current.id_p = g$ *and* $established[g]_p = false$, *then there is no* $x \in allstate[p, g]$ *with* $x.high = g$.

Proof. Lemma 6.10 implies that $highprimary_p < g$.

We prove the statement by induction. $newview_p$ is the only action that can convert the hypothesis from false to true, and it guarantees the conclusion, by Lemma 6.11 applied to the pre-state.

To convert the conclusion from true to false, we would have to end the step with $highprimary_p = g$ (since the other pieces of $allstate[p]$ are derived from p's state). But this doesn't happen, by the claim at the beginning of the proof. □

Lemma 6.15 *The following is true in all reachable states of VStoTO-system.*
If $x \in allstate[p, g]$ *then there exists* $v \in created$ *and* $q \in v.set$ *such that*

1. $x.high = v.id$
2. $established[x.high]_q$
3. $x.ord = buildorder[q, x.high]$.
4. *either* $x.high = g$ *or* $current.id_q > v.id$

Proof. The proof is by induction, so consider a step in which the state changes from s to s' by the action π. If $x \in s'.allstate[p, g]$, then in most cases, there is $y \in s.allstate[p, g]$ with $y.high = x.high$ and $y.ord = x.ord$, to which we apply the induction hypothesis. The only cases where this does not happen are as follows:

- π is the receipt by p of an ordinary message in a primary view, and x is the summary whose components are taken from the state of p. In this case we take $v = g$ and $q = p$.
- π is the establishment of a new primary view g at p, and x is the summary whose components are taken from the state of p. In this case we take $v = g$ and $q = p$.
- π is **newview**$(v)_p$, where $v.id = g$, and x is the summary whose components are taken from the state of p. In this case, there is $y \in s.allstate[p, s.current_p]$ with $y.high = x.high$ and $y.ord = x.ord$, to which we apply the induction hypothesis.

□

Lemma 6.16 *The following is true in all reachable states of VStoTO-system.*
If $v \in created$ *and* $established[v.id]_p$ *then for every* $q \in v.set$, $current.id_q \geq v.id$.

Proof. When $established[v.id]_p$ first becomes true, the code for **gprcv** shows that $domain(gotstate_p) = v.set$, so $allstate[q, v.id]$ is non-empty for all $q \in v.set$. Part 4 of Lemma 6.6 thus implies $current_q \geq v.id$. This is maintained inductively in all later states, by the monotonicity of $current_q$. □

The following is a key invariant; it can be used to show that information from certain processors' tentative orders for a primary view v is also present in all summaries with higher viewids. The hypothesis says that every processor in $v.set$ that has a $current.id$ higher than $v.id$ has succeeded in establishing v, and moreover, has succeeded in including the sequence σ in its *order* for view v. The conclusion says that anyplace in the state where information about a higher view than v is present, information about σ is also present.

Lemma 6.17 *The following is true in all reachable states of VStoTO-system.*
Suppose that $v \in created$, *v.set contains a quorum,* $\sigma \in L^\star$, *and for every* $p \in v.set$, *the following is true:*
If $current.id_p > v.id$ *then* $established[v.id]_p$ *and* $\sigma \leq buildorder[p, v.id]$.

Then for every $x \in allstate$ *with* $x.high > v.id$, $\sigma \leq x.ord$.

Proof. The statement is vacuously true if $v \notin created$.

Otherwise argue by induction, where s denotes the state before a step and s' the state afterwards.

If $v \in s'$, *created* and $v \notin s.created$, then the action involved must be **createview**(v). In this case, we claim that the conclusion is true because no $x \in allstate$ has $x.high > v.id$. To see this, fix $x \in s'.allstate$, say, $x \in s'.allstate[p]$. Then Lemma 6.11 implies that $x.high \leq s'.current.id_p$. Lemma 6.1 implies that $s'.current_p \in s'.created$. And the code for **createview**(v) implies that $v.id$ is the largest id in $s'.created$, in particular, that $v.id \geq s'.current.id_p$. So $x.high \leq v.id$.

So for the rest of the argument, we fix v and assume that $v \in s.created$. Also fix σ.

As usual, the interesting steps are those that convert the hypothesis from false to true, and those that keep the hypothesis true while converting the conclusion from true to false.

In this case, there are no steps that convert the hypothesis from false to true: If there is some $p \in v.set$ for which $s.current.id_p > v.id$ and either $s.established[v.id]_p = false$ or σ is not a prefix of $buildorder[p, v.id]$, then also $s'.current.id_p > v.id$ (the id never decreases) and either $s.established[v.id]_p = false$ or σ is not a prefix of $buildorder[p, v.id]$. (These two cases carry over, since $s.current.id_p > v.id$ implies that $established[v.id]_p$ and $buildorder[p, v.id]$ cannot change.)

So it remains to consider any steps that keep the hypothesis true while converting the conclusion from true to false. So suppose that $x \in s'.allstate$ and $x.high > v.id$. If also $x \in s.allstate$ then we can apply the inductive hypothesis, which implies that $\sigma \leq x.ord$, as needed. So the only concern is with steps that produce a new summary.

Any step that produces the new summary x by modifying an old summary $x' \in s.allstate$, in such a way that $x'.ord \leq x.ord$ and $x'.high = x.high$, is easy to handle: For such a step, $x'.high > v.id$ and so the inductive hypothesis implies that $\sigma \leq x'.ord \leq x.ord$, as needed. So the only concern is with $\mathbf{gprcv}_p$ steps that deliver the last state-exchange message to some process p.

If the $\mathbf{gprcv}_p$ is not for a primary, then the new summary x that is produced, in p's state, takes its *highprimary* and *order* values directly from some summary x' which is in the range of $s'.gotstate_p$. By the code, such an x' is either in the range of $s.gotstate_p$, or else it is the summary whose receipt is the step we are considering. In either of these cases, $x' \in s.allstate$, so the inductive hypothesis yields the result.

This leaves the case where $\mathbf{gprcv}_p$ establishes a primary w, and x is the summary composed of the new values of the state components of p. Thus $x.high = w.id$. Let x' be the summary of $q' = chosenrep$ in state s.

We claim that $x'.high \geq v.id$. To see the claim, fix any element q'' in $w.set \cap v.set$; such a q'' must exist, because each contains a quorum. Recall that the condition for establishing a primary shows $domain(s'.gotstate_p) = w.set$, so by the code, either $q'' \in domain(s'.gotstate_p)$, or else q'' is the sender of the message whose receipt is the step we are examining. In the former case, let x'' be the summary $s.gotstate(q'')_p$; in the latter let x'' be the summary whose receipt is the event. In either case we have $x'' \in s.allstate[q'', w.id]$. Thus, part 4 of Lemma 6.6 implies that $current.id_{q''} \geq w.id$. We have that $x.high > v.id$ by assumption, and $x.high = w.id$ by the code; therefore, $w.id > v.id$. So also $current.id_{q''} > v.id$.

Recall that we are in the case where the hypothesis of this lemma is true. Therefore, by this hypothesis, we obtain that $established[v.id]_{q''}$ and $\sigma \leq buildorder[q'', v.id]$. By Lemma 6.13, (applied with q'' replacing p) we obtain $x''.high \geq v.id$. By the definition of q' as a member that maximizes the *high* component in the summary recorded in *gotstate*, we have $x'.high \geq x''.high$. Therefore $x'.high \geq v.id$, completing our demonstration of the claim.

If $x'.high > v.id$ then we can apply the induction hypothesis to x' and we are done, since $x'.ord \leq x.ord$. So suppose $x'.high = v.id$. Note that $x' \in allstate[q', w.id]$. By Lemma 6.15, there must exist[4] $q \in v.set$ so that $established[v.id]_q$, $x'.ord = buildorder[q, v.id]$, and (either $x'.high = w.id$ or $current.id_q > v.id$). Since $x'.high = v.id < x.high = w.id$, the last property can be simplified to $current.id_q > v.id$. The hypothesis of this lemma says that this forces $\sigma \leq buildorder[q, v.id]$. Since $x'.ord \leq x.ord$ by the code for this event, and $x'.ord = buildorder[q, v.id]$ as shown above, this is what we need. □

The invariant given in the corollary implies that once all members if a primary view agree on a common part of the tentative order, all processors in a higher view will also share that sequence in that order.

Corollary 6.18 *The following is true in all reachable states of VStoTO-system.*
Suppose that $v \in created$, *v.set contains a quorum,* $\sigma \in L^\star$, *and for every* $p \in v.set$, $established[v.id]_p$ *and* $\sigma \leq buildorder[p, v.id]$.
Then for every $x \in allstate$ *with* $x.high \geq v.id$, $\sigma \leq x.ord$.

[4] Direct application of the Lemma actually shows the existence of some $\hat{v}$ and $q \in \hat{v}.set$, but since $x'.high = \hat{v}.id$ and also $x'.high = v.id$, uniqueness of viewids shows we may take $\hat{v}$ to be v itself.

Proof. If $x.high > v.id$, then we can apply Lemma 6.17, since the premise of this Corollary deals with every $p \in v.set$, and therefore is stronger than the premise of Lemma 6.17, which only covers those p where $current.id_p > v.id$.

When $x.high = v.id$, we apply apply Lemma 6.15 to x, which gives $v' \in created$ and $q' \in v'.set$ such that $x.high = v'.id$, $established[x.high]_{q'}$, and $x.ord = buildorder[q', x.high]$. Since $v.id = v'.id$, Lemma 4.1 shows $v = v'$. Substituting in the facts above we see $x.ord = buildorder[q', v.id]$. Since $q' \in v.set$, we can apply the premise of the corollary to see that $\sigma \leq buildorder[q', v.id]$; that is, $\sigma \leq x.ord$, as required. □

The next lemma makes precise the fact that a label is in $safe\text{-}labels_p$ only after it (and all prior labels in $order_p$) were placed in $order_q$ at every member q of $current.set_p$

Lemma 6.19 *If $l \in safe\text{-}labels_p$ and σ is a prefix of $order_p$ that is terminated by l, then $primary_p$ and for all $q \in current.set_p$, $\sigma \leq buildorder[q, current.id_p]$*

The next lemma shows that in any summary, the *ord* component is closed under the relation "sent-before-by-one-client".

Lemma 6.20 *The following is true in all reachable states of VStoTO-system.*
Suppose $l, l' \in L$ and $i \in \mathcal{N}^{>0}$. If $l, l' \in domain(allcontent)$ and $l.origin = l'.origin$ and $l < l'$ and $x \in allstate$ and $l' = x.ord(i')$ then there exists i such that $i < i' \wedge l = x.ord(i)$

Next we show that $x.confirm$ is a prefix of a known sequence. This leads to consequences that show the consistency of the confirmed sequence of labels at different places in the system.

Lemma 6.21 *The following is true in all reachable states of VStoTO-system. If $x \in allstate$ then*

1. *There exists $v \in created$ such that $v.id \leq x.high$, $v.set$ contains a quorum, and for every $q \in v.set$, $established[v.id]_q$ and $x.confirm \leq buildorder[q, v]$.*
2. *$x.next \leq length(x.ord) + 1$*

Remark: an immediate consequence of part (2) is that $length(x.confirm) + 1 = x.next$.

Proof. The strategy is to show that (1) and (2) hold in the post-state, by induction, using (1) and (2) from the pre-state.

How is the post-state proved: In the step from s to s', in most cases, there is y in $s.allstate$ so that $y.next = x.next$, $y.ord = x.ord$ (and hence $y.confirm = x.confirm$) and also $y.high = x.high$. If this holds induction hypothesis gives us what we want, since $buildorder[q, v]$ increases monotonically through an execution.

The places where a problem might happen are the following.

- **confirm**$_p$. If x is not the summary from the state of p, the inductive argument works. If x is the summary from the state of p, the precondition of the event is that the newly confirmed message has label in $safe\text{-}set_p$, so Lemma 6.19 shows that we have (1) with taking v to be $current_p = s'.high$. The precondition also gives $(x.next - 1) \in domain(x.ord)$, thus showing (2).
- **gprcv**$(\langle l, a\rangle)_p$ For a summary other than that from the state of p, the inductive argument applies. Where x is the summary from the state of p, let y denote the summary in the pre-state taken from the state of p. The code shows that $x.high = y.high$, $x.next = y.next$, and $x.ord$ is an extension of $y.ord$. By (2) applied to x', we see that $y.next \leq length(y.ord) + 1$ and therefore $x.next \leq length(x.ord) + 1$. This is (2) applied to x; also it shows that $x.confirm = y.confirm$, so that the inductive hypothesis of (1) applied to y gives (1) applied to x.
- receipt of the final state exchange message at p. For a summary other than that from the state of p, the inductive argument applies. Where x is the summary from the state of p, let w denote the summary, among those in $gotstate_p$ after the action, with the highest value for $w.next$. The code shows that $x.next = w.next$. Now w is in $allstate$ in the pre-state (it is either in $gotstate$, or else it is

the summary received in the final state-echange message, in which case it is in the *queue* component of *VS-machine*). The inductive hypothesis shows that $w.confirm$ has length $w.next - 1$, and that there is $v \in created$ such that $v.id \leq w.high$ and $\forall q \in v.set.$ $(established[v]_q \wedge w.confirm \leq buildorder[q, v])$. Now let z denote the summary of $chosenrep(gotstate)$, as calculated in the effect of the action. Since $z.high \geq w.high \geq v.id$ (recall the definition of z as being from a representative, that is, having maximal highprimary among summaries in *gotstate*), Corollary 6.18 shows that $w.confirm \leq z.ord$. Since $z.ord \leq x.ord$ by the code (whether the newly established view is primary or not), we deduce that $w.confirm$ is a prefix of $x.ord$; as $length(w.confirm) = w.next - 1 = x.next - 1$, we have $x.confirm = w.confirm$. Also by the code (for a non-primary view) or Lemma 6.11 (for a primary view) we have $x.high \geq w.high$. Thus the inductive hypothesis applied to w gives (1) and (2) for x.

□

Corollary 6.22 *The following is true in all reachable states of VStoTO-system.*
If $x_1, x_2 \in allstate$ and $x_1.high \leq x_2.high$, then $x_1.confirm \leq x_2.ord$.

Proof. This is done by using lemma 6.21 (part (1)) with $x = x_1$, giving v with $v.id \leq x_1.high$ and $x_1.confirm \leq buildorder[q, v]$. Now the hypothesis of Corollary 6.18 applies for $\sigma = x_1.confirm$; since $x_2.high \geq v.id$ the conclusion of that Lemma holds for x_2, that is $x_1.confirm \leq x_2.ord$. □

Corollary 6.23 *The following is true in all reachable states of VStoTO-system.*
For any $x, x' \in allstate$, either $x.confirm \leq x'.confirm$ or $x'.confirm \leq x.confirm$.

Proof. Wlog $x.high \leq x'.high$. From lemma 6.22, we have that both $x.confirm$ and $x'.confirm$ are prefixes of $x'.order$. □

Invariant 6.23 allows us to define another derived variable that represents the collective knowledge of the confirmed order, throughout the system. Namely, in any reachable state, we write *allconfirm* for $lub_{x \in allstate}(x.confirm)$.

6.2 Simulation Relation

Next, we define the simulation relation f. We define it as a function from reachable states of *VStoTO-system* to states of *TO-machine*. (We assume an arbitrary default value for unreachable states.) Namely, if x is a reachable state of *VStoTO-system*, then $f(x) = y$ where:

1. $y.queue = applyall(\langle x.allcontent, origin \rangle, x.allconfirm)$,
   where the selector *origin* is regarded as a function from labels to processors.
2. $y.next[p] = x.next\text{-}report_p$.
3. $y.pending[p] = applyall(x.allcontent, s)$ where s is the sequence of labels such that
   (a) $range(s)$ is the set of labels l such that $l.origin = p$, $\langle l, a \rangle \in x.allcontent$ for some a, and $l \notin range(allconfirm)$.
   (b) s is ordered according to the label order.

The first clause says that $y.queue$ is the sequence of $\langle value, origin \rangle$ pairs corresponding to the sequence $x.allconfirm$ of labels that are confirmed anywhere in the system. For each label in $x.allconfirm$, the set $x.allcontent$, which contains all the content information that appears anywhere in the system, is used to obtain the value, and *origin* is used to extract the origin. (Note that the set of pairs $x.allcontent$ is treated as a function, and that the two functions are paired together into one for use with the *applyall* operator.) The second clause defines $y.next[p]$ directly from the corresponding next-pointer in x. The third clause defines $y.pending[p]$ to be the sequence of values corresponding to all the labels in the system with origin p that are not included in $x.allconfirm$, arranged in label order. For each such label, $x.allcontent$ is used to obtain the value. Note that the well-definedness of this simulation rests on the invariant that says that $x.allcontent$ is a function, and on Invariant 6.23, which yields the definedness of *allconfirm*.

Lemma 6.24 *Function f is a forward simulation from VStoTO-system to TO-machine.*

Proof. The correspondence in the initial state is trivial. So consider any step (x, π, x') of *VStoTO-system*, and $y = f(x)$. We argue depending on the action involved in π.

$\pi = \mathbf{bcast}(a)_p$ Since π is an input to *TO-machine*, π is enabled in y. Now the effect of π shows that $x'.allconfirm = x.allconfirm$, and $x'.allcontent$ is the union of $x.allcontent$ with $\langle l, a\rangle$ where $l = \langle x.current_p, x.nextseqno_p, p\rangle$; by Lemma 6.3, this new label l is greater than all labels already in the domain of *allcontent*, so $f(x').pending[p]$ is formed by adding a to the end of $f(x).pending[p]$, thus showing that $(f(x), \pi, f(x'))$ is a step of *TO-machine*.

$\pi = \mathbf{confirm}_p$ Clearly the effect of π shows $x.allcontent = x'.allcontent$.

If $x.nextconfirm_p \leq length(x.allconfirm)$ then Lemma 6.23 and the effect of π shows that $x'.allconfirm = x.allconfirm$, so that $f(x) = f(x')$.

Otherwise $x.nextconfirm_p = length(x.allconfirm) + 1$, so the effect of π shows that $x'.allconfirm = x.allconfirm \cdot \langle l\rangle$ where $l = x.order_p(x.nextorder_p)$. Let $q = l.origin$ and $a = x.allcontent(l)$. We claim that $(f(x), \mathbf{to\text{-}order}(a, q), f(x'))$ is a step of *TO-machine*.

We first show that $\mathbf{to\text{-}order}(a, q)$ is enabled in $f(x)$. We have $l \in domain(x.allcontent)$ and $l \notin setof(x.allconfirm)$; this means that a is an element of the sequence $f(x).pending[q]$. Also by Lemma 6.20, any lower label with origin q is in $x.confirm_p$ and so in $x.allconfirm$. Since the sequence S used to define $f(x).pending[q]$ is arranged by label, we see that l is the head of S, and so a is the head of $f(x).pending[q]$, as required. Further, the equation above for $x'.allconfirm$ shows that $f(x').queue = f(x).queue \cdot \langle\langle\langle a, p\rangle\rangle\rangle$, and this is what is needed to show that π takes $f(x)$ to $f(x')$.

$\pi = \mathbf{gprcv}(s)_{p,q}$ In some cases this may change the value of $nextconfirm_q$, but in every situation it leaves *allconfirm* unchanged (it only moves $nextconfirm_q$ to a value already somewhere in *allstate*) Thus $f(x') = f(x)$.

$\pi = \mathbf{brcv}(a)_{pq}$ We need to show that π is enabled in $f(x)$ as an action of *TO-machine*, but this is immediate from the fact that π is enabled in x as an action of *VStoTO*. Similarly, the effect corresponds (only $nextreport_q$ is altered).

Other actions The other actions leave $f(x') = f(x)$.

□

Theorem 6.25 *Every trace of VStoTO-system is a trace of TO-machine.*

7 Performance and Fault-Tolerance

We argue that the performance and fault-tolerance characteristics of *TO* (for certain values of the parameters) are implied by the corresponding ones for *VS* (for certain parameter values), together with performance and fault-tolerance characteristics of the *VStoTO* processes. In order to do this, we need a richer model for the system than we have been using so far. This richer model must include timing and failure information. We define this richer model in two separate pieces, for *VStoTO* and for *VS*.

For the *VStoTO* part, we define a timed automaton $VStoTO'_p$ for every p. This timed automaton is obtained by modifying the untimed automaton $VStoTO_p$ as follows:

- Add new input actions $\mathbf{good}_p$, $\mathbf{bad}_p$ and $\mathbf{ugly}_p$.
- Add new time-passage actions $\nu(t)$ for all $t \in \mathcal{R}^{>0}$.
- Add a new state component *failure-status*, with values in $\{good, bad, ugly\}$, initially *good*.
- Add new code fragments for the failure status actions, just setting the failure-status variable appropriately.
- Add a new precondition to each output and internal action, that $failure\text{-}status \neq bad$.
- Add a code fragment for each $\nu(t)$:

$\nu(t)$
Precondition:
  if *failure-status* = *good* then
    no output or internal action is enabled
Effect:
  none

The new precondition on output and internal actions says that the processor takes no steps when its failure status is *bad*. The new time-passage actions are allowed to happen at any point, unless there is some output or internal action that is supposed to happen immediately (because it is enabled and the processor is *good*).

For the *VS* part, we now fix b and d to be particular constants. We assume that we have any timed automaton A that satisfies the specification $VS(b,d,Q)$ from Section 4 for every set Q of processors that contains a quorum.

Define *VStoTO′-system* to be the composition of A and $VStoTO'_p$ for all $p \in P$, with the actions used for communication between the two layers (that is, the **gpsnd**, **gprcv**, **safe** and **newview**), hidden. Note that the failure status input actions are not hidden. The composition operator used here is timed automaton composition.

We show that any admissible timed trace of *VStoTO′-system* satisfies *TO-property*, for certain values of the parameters:

Theorem 7.1 *Every admissible timed trace of VStoTO′-system satisfies TO-property$(b+d, d, Q)$ for every Q that contains a quorum.*

Proof. Let (β, ∞) be any admissible timed trace of *VStoTO′-system*, and let α be an admissible timed execution of *VStoTO′-system* that gives rise to β. Fix Q to be any set of processors containing a quorum.

We first show Condition 1 of the definition of *TO-property*, that β with the timing information removed is a trace of *TO-machine*. This follows from general composition results for timed automata (see, e.g., Chapter 23 of [20]), using what we have already proved in the safety part of the paper.

In more detail, regard α as a timed execution of the composed system composed of $VStoTO'$ and A, without the interface actions being hidden. Then project α on $VStoTO'$ and A to give timed executions α_1 of $VStoTO'$ and α_2 of A, respectively – this uses a projection result for timed automata. Removing the timing information from the timed trace of α_1 yields a trace of *VStoTO*, by definition of $VStoTO'$. The first part of *VS-property* implies that removing the timing information from the timed trace of α_2 yields a trace of *VS-machine*. Now paste these two timed traces together, using a pasting lemma for composition of untimed automata, to yield a timed trace β_1 of the composition of *VStoTO* and *VS-machine*. We claim that β_1 restricted to the external actions of *TO-machine* is equal to the original trace β. But then β is a trace of *VStoTO-system*, and so by Theorem 6.25 is a trace of *TO-machine*.

The more interesting property to show is Condition 2, the performance and fault-tolerance property. Our strategy for proving the needed property of β is to use an auxiliary "conditional" property *VStoTO-property* of α. *VStoTO-property* uses the "conclusion" part of *VS-property*(b,d,Q) for A, together with the performance and fault-tolerance assumptions for the processors $VStoTO'_p$, to infer the conclusion part of *TO-property*$(b+d,d,Q)$.

VStoTO-prop:
Suppose that α can be written as $\alpha'\alpha''$, such that:
1. α'' contains no **newview** events at locations in Q.
2. The latest views at all locations in Q after α' are the same, say $\langle g, S\rangle$, where $S = Q$.
3. Every message sent from a location in Q in α while in view $\langle g, S\rangle$ at time t has corresponding **safe** events at all members of Q by time $\max(t, ltime(\alpha')) + d$.
4. α'' contains no failure status events for locations in Q or for pairs including a location in Q.
5. All locations in Q and all pairs of locations in Q are *good* after α'.
6. If $p \in Q$ and $q \notin Q$ then (p,q) is *bad* after α'.

Then α'' can be written as $\alpha'''\alpha''''$, where
1. $ltime(\alpha''') \le d$

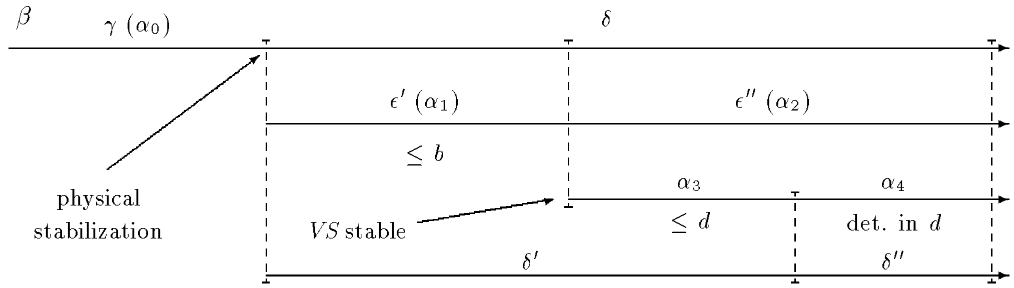

Figure 6: Performance argument diagram

2. Every data value sent from a location in Q in α at time t is delivered at all members of Q by time $\max\{t, ltime(\alpha'\alpha''')\}+d$.
3. Every data value delivered to any location in Q at time t is delivered at all members of Q by time $\max\{t, ltime(\alpha'\alpha''')\}+d$.

We prove *VStoTO-property* operationally. In our proof, the execution fragment α''' whose existence is asserted in the conclusion of *VStoTO-property* extends until every member of Q has received the safe indication for every state-exchange message sent in view $\langle g, S\rangle$. Our proof uses the fact that Q contains a quorum, and also the fact that the "good" processors perform enabled actions immediately.

Based on *VStoTO-property*, it is easy to unwind the definitions and prove that the complete system satisfies *TO-property*$(b+d, d, Q)$. Suppose that $(\beta,\infty) = (\gamma, l)(\delta,\infty)$ is an admissible timed trace of *VStoTO$'$-system*. Suppose that all the following hold:

1. δ contains no failure status events for locations in Q or for pairs including a location in Q.
2. All locations in Q and all pairs of locations in Q are *good* after γ.
3. If $p \in Q$ and $q \notin Q$ then (p, q) is *bad* after γ.

We show that (δ,∞) can be written as $(\delta', l')(\delta'',\infty)$, where

1. $l' \leq b + d$.
2. Every data value sent from a location in Q in β, say at time t, is delivered at all members of Q by time $\max(t, l + l') + d$.
3. Every data value delivered to any location in Q, say at time t, is delivered at all members of Q by time $\max(t, l + l') + d$.

By the definition of *VS-property*(b, d, Q), we have that (δ,∞) can be written as $(\epsilon', t')(\epsilon'', t'')$, where

1. $t' \leq b$.
2. No *newview* events occur in ϵ'' at processors in Q.
3. The latest views at all locations in Q after $\gamma\epsilon'$ are the same, say $\langle g, S\rangle$, where $S = Q$.
4. Every message sent from a location in Q in β while in view $\langle g, S\rangle$, say at time t, has corresponding *safe* events at all members of Q by time $\max(t, l + t') + d$.

Next, we apply the conditional property to the timed execution α that gives rise to the timed trace (β,∞). Let α_0, α_1 and α_2 be the parts of α that give rise to γ, ϵ' and ϵ'', respectively.

The conditional property implies that α_2 can be written as $\alpha_3\alpha_4$, where

1. $ltime(\alpha_3) \leq d$
2. Every data value sent from a location in Q in α, say at time t, is delivered at all members of Q by time $\max(t, ltime(\alpha_0\alpha_1\alpha_3)) + d$.
3. Every data value delivered to any location in Q, say at time t, is delivered at all members of Q by time $\max(t, ltime(\alpha_0\alpha_1\alpha_3)) + d$.

Then we claim that taking δ' to be the timed trace of $\alpha_1\alpha_3$ and δ'' to be the timed trace of α_4 yields the needed properties. To see that $l' \leq b + d$, note that $l' = ltime(\alpha_1) + ltime(\alpha_3) \leq b + d$. For the delivery times, the conclusion of the conditional property provides bounds in terms of $ltime(\alpha_0\alpha_1\alpha_3)$, which is the same as $l + l'$, which is as needed. □

As a consequence of Theorem 7.1, we have the main result:

Theorem 7.2 *VStoTO'-system satisfies the specification $TO(b + d, d, Q)$, for every Q that contains a quorum.*

8 Implementing *VS*

In this paper we do not offer a formal proof that *VS* can be implemented. Instead, we sketch one implementation, informally. The implementation is based on the 3-round membership protocol[5] given by Cristian and Schmuck in [10]. In this protocol, once a view is formed, it is "held together" by a circulating token, which is started by a deterministically chosen leader, and travels from member to member around a logical ring. Each processor knows the size of the ring, and so it sets a timer that expires if the token does not return in reasonable time. If a member crashes, or communication failure causes the token to be lost or delayed, the timer expiration triggers formation of a new view. Similarly a new view is initiated if contact occurs from a processor outside the current membership.

Once a processor determines that a new view is needed, it broadcasts a call-for-participation in the new view (together with a unique viewid chosen to be larger than any the processor has seen). The membership of the view is all processors that reply to the broadcast. A processor may not reply to one call after replying to another with higher viewid. Once the membership is determined, this is sent to the members which then join the view (unless they have already agreed to participate in a view with higher viewid). A leader within the view membership launches the token.

To provide ordered message delivery, we use the token to carry the sequence of messages. Each processor buffers messages from the client until the token passes; the messages are then appended to the token. Each processor examines the sequence carried by the token, and passes to its client any messages that it has not already passed on. The token also carries an indication of how many messages each member passed to its client, when the token last left that member. This is the basis for the safe indication: a message is safe once the token records that all members have passed it to the corresponding clients.

Suppose the following hold of the underlying physical system of processors and links:

- While $status_p = good_p$, processor p takes any enabled step immediately.
- While $status_p = bad$, processor p takes no locally controlled step.
- While $status_{pq} = good$, every packet sent from p to q arrives within time δ
- While $status_{pq} = bad$, no packet is delivered from p to q

As analyzed in [10] the protocol above implements $VS(b, d, Q)$, where Q is any set of processors, $b = 9\delta + \max\{\pi + (n + 3)\delta, \mu\}$, and $d = 2\pi + n\delta$. Here, n is the number of processors in Q, π is the spacing of token creation by the ring leader (this must satisfy $\pi > n\delta$), and μ is the spacing of attempts to contact newly connected processes.

Some remarks about a correctness proof for this implementation: The safety claim involves showing that any trace of the implementation is a trace of *VS-machine*. Since traces include only external events (and not internal events like createview), the implementation needn't preserve the order of createview events (in fact, the implementation needn't even have createview events).

To show this trace inclusion, we use *WeakVS-machine*. We first show that *WeakVS-machine* implements *VS-machine*, in the sense of trace inclusion, provided that the viewid set G does not contain an

[5] A different implementation could use the one-round protocol of [10]. However, this would stabilize less quickly.

infinite number of elements smaller than any particular element g. This proof is achieved by reordering createview events, pushing any such event earlier than any createview event for a bigger view.

Then use a forward simulation to show that the algorithm implements *WeakVS-machine*. This forward simulation should be straightforward, mapping to **createview** in *WeakVS-machine* the event in C-S where a processor defines the membership of the view, after waiting 2δ units since sending the "newgroup" message (the membership is the set that sent "accept" responses). Uniqueness of viewids is immediate since in CS they have a procid as low-order part (and a stable seqno as highorder part). Note that we still have monotonicity on the viewids that p sees, because $\mathbf{newview}(v)_p$ still has precondition that $v.id > current.id_p$.

An operational argument should work for performance and fault-tolerance.

9 Conclusions and discussion

The construction of distributed applications is substantially aided by the availability of distributed system building blocks, such as message passing, multicast or remote procedure call. Some sophisticated application are most effectively aided by the availability of building blocks providing higher level functions and guarantees, such as universally ordered broadcast. In order for a building block to be useful, it must be precisely specified, the specification must be as simple as posisble, the correctness and performance quarantees must be explicitly stated, last but not least, the building block must be implementable.

We presented a simple specification for a partionable group communication service. We demonstrated the utility of the service by using it in specifying and proving correct a total order messaging service. The performance and fault-tolerance properties of the total order service are derived from the performance and fault-tolerance properties of the group communication service. We also described one implementation of the service.

Future work involves using *VS* to construct other applications, for example, load-balancing applications. Considering other applications may lead to different variants of the specification; it would be interesting to identify these variants and understand how they relate to each other. It also remains to apply the approach of this paper to the task of specifying and analyzing other group-communication services, e.g., services involving multiple groups with possibly-overlapping memberships, services in which processors voluntarily join or leave groups, or services that include combined broadcasts and convergecasts. groups, or services that include combined broadcasts and convergecasts.

Acknowledgments We thank Ken Birman, Tom Bressoud, Danny Dolev, Brad Glade, Idit Keidar, Debby Wallach, and especially Dalia Malki for discussions about practical aspects of group communication services. Myla Archer has mechanically checked some of the invariants using PVS, thereby helping us to debug and polish the proofs. Roger Khazan contributed several improvements to the formal models. Roberto De Prisco and Nicole Lesley made several helpful suggestions.

This research was supported by the following contracts: ARPA F19628-95-C-0118, AFOSR-ONR F49620-94-1-0199, U.S. Department of Transportation: DTRS95G-0001- YR. 8, and NSF 9225124-CCR.

References

[1] Y. Amir, D. Dolev, P. Melliar-Smith and L. Moser, “Robust and Efficient Replication Using Group Communication” Technical Report 94-20, Department of Computer Science, Hebrew University., 1994.

[2] Y. Amir, L. Moser, P. Melliar-Smith, D. Agrawal and P. Ciarfella, “Fast Message Ordering and Membership Using a Logical Token-Passing Ring”, in *Proc. of IEEE International Conference on Distributed Computing Systems*, 1993, pp 551–560.

[3] O. Babaoglu, R. Davoli, L. Giachini and M. Baker, “Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems”, in *Proc. of Hawaii International Conference on Computer and System Science*, 1995, volume II, pp 612–621.

[4] O. Babaoglu, R. Davoli and A. Montresor, “Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems”, Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, Italy.

[5] O. Babaoglu, R. Davoli, L. Giachini and P. Sabattini, "The Inherent Cost of Strong-Partial View Synchronous Communication", in *Proc of Workshop on Distributed Algorithms on Graphs*, pp 72–86, 1995.

[6] K.P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, Los Alamitos, CA, 1994.

[7] T.D. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost, "On the Impossibility of Group Membership", in *Proc. of 15th Annual ACM Symp. on Princ. of Distr. Comput.*, pp. 322-330, 1996.

[8] F. Cristian, "Synchronous and Asynchronous Group Communication", *Comm. of the ACM*, vol. 39, no. 4, pp. 88–97, 1996.

[9] F. Cristian, "Group, Majority and Strict Agreement in Timed Asynchronous Distributed Systems", in *Proc. of 26th Conference on Fault-Tolerant Computer Systems*, 1996, pp. 178–187.

[10] F. Cristian and F. Schmuck, "Agreeing on Processor Group Membership in Asynchronous Distributed Systems", Technical Report CSE95-428, Department of Computer Science, University of California San Diego.

[11] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communications", *Comm. of the ACM*, vol. 39, no. 4, pp. 64–70, 1996.

[12] D. Dolev, D. Malki and R. Strong "A Framework for Partitionable Membership Service", Technical Report TR94-6, Department of Computer Science, Hebrew University.

[13] P. Ezhilchelvan, R. Macedo and S. Shrivastava "Newtop: A Fault-Tolerant Group Communication Protocol" in *Proc. of IEEE International Conference on Distributed Computing Systems*, 1995, pp 296–306.

[14] A. Fekete, F. Kaashoek and N. Lynch "Providing Sequentially-Consistent Shared Objects Using Group and Point-to-point Communication" in *Proc. of IEEE International Conference on Distributed Computer Systems*, 1995, pp 439–449.

[15] R. Friedman and R. van Renesse, "Strong and Weak Virtual Synchrony in Horus", Technical Report TR95-1537, Department of Computer Science, Cornell University.

[16] M. Hiltunen and R. Schlichting "Properties of Membership Services", in *Proc. of 2nd International Symposium on Autonomous Decentralized Systems*, pp 200–207, 1995.

[17] F. Jahanian, S. Fakhouri and R. Rajkumar, "Processor Group Membership Protocols: Specification, Design and Implementation" in *Proc. of 12th IEEE Symposium on Reliable Distributed Systems* pp 2–11, 1993.

[18] I. Keidar and D. Dolev, "Efficient Message Ordering in Dynamic Networks", in *Proc. of 15th Annual ACM Symp. on Princ. of Distr. Comput.*, pp. 68-76, 1996.

[19] L. Lamport, "Time, Clocks. and the Ordering of Events in a Distributed System, *Comm. of the ACM*, vol. 21, no. 7, pp. 558-565, 1978.

[20] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[21] N.A. Lynch and M.R. Tuttle, "An Introduction to Input/Output Automata", *CWI Quarterly*, vol.2, no. 3, pp. 219-246, 1989.

[22] N.A. Lynch and F. Vaandrager, "Forward and Backward Simulations — Part I: Untimed Systems", *Information and Computation*, vol. 121, no. 2, pp. 214-233, 1995.

[23] N.A. Lynch and F. Vaandrager, "Forward and backward simulations – Part II: Timing-based systems", *Information and Computation* vol. 128, no. 1, pp 1-25, 1996.

[24] L. Moser, Y. Amir, P. Melliar-Smith and D. Agrawal, "Extended Virtual Synchrony" in *Proc. of IEEE International Conference on Distributed Computing Systems*, 1994, pp 56–65.

[25] L.E. Moser, P.M. Melliar-Smith, D.A. Agarawal, R.K. Budhia and C.A. Lingley-Papadopolous, "Totem: A Fault-Tolerant Multicast Group Communication System", *Comm. of the ACM*, vol. 39, no. 4, pp. 54-63, 1996.

[26] G. Neiger, "A New Look at Membership Services", in *Proc. of 15th Annual ACM Symp. on Princ. of Distr. Comput.*, pp. 331-340, 1996.

[27] R. van Renesse, K.P. Birman and S. Maffeis, "Horus: A Flexible Group Communication System", *Comm. of the ACM*, vol. 39, no. 4, pp. 76-83, 1996.

[28] A. Ricciardi, "The Group Membership Problem in Asynchronous Systems", Technical Report TR92-1313, Department of Computer Science, Cornell University.

[29] A. Ricciardi, A. Schiper and K. Birman, "Understanding Partitions and the "No Partitions" Assumption", Technical Report TR93-1355, Department of Computer Science, Cornell University.

[30] F. Schneider, "Implementing Fault-Tolerant Services using the State machine Approach: A Tutorial", *ACM Computing Surveys*, vol. 22, no. 4, 1990.