Specifying and Using a Partitionable Group Communication Service

Alan Fekete* Nancy Lynch† Alex Shvartsman†

Abstract

A new, simple formal specification is presented for a partitionable view-oriented group communication service. The specification consists of a state machine to express safety requirements and a timed trace property to express performance and fault-tolerance requirements. The specification is used to construct a totally-ordered-broadcast application, using an algorithm (based on algorithms of Amir, Dolev, Keidar and others) that reconciles information derived from different views of the group. Correctness of the resulting application is proved, and its performance and fault-tolerance analyzed. The specification has a simple implementation, based on a group membership algorithm of Cristian and Schmuck.

1 Introduction

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. 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 serious issue.

In this paper, 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]

*Basser Department of Computer Science, Madsen Building F09, University of Sydney, NSW 2006, Australia.

†MIT Laboratory for Computer Science, 545 Technology Square, NE43-365, Cambridge, MA 02139, USA.

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

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 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*. *TO* serves as the correctness definition for the ordered-broadcast application. It consists of 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*. *VS* includes 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* service also provides a "safe" indication, once a message has been delivered to all members of the view.

The most important differences between *VS* and other group communication specifications are:

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 is 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.

The differences represented by points 2, 4 and 6 mean that *VS* 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, Keidar, Melliar-Smith and Moser [18, 1]. These 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

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

[3] The "replicated state machine" approach of Lamport [19], surveyed by Schneider in [30], is one such approach.

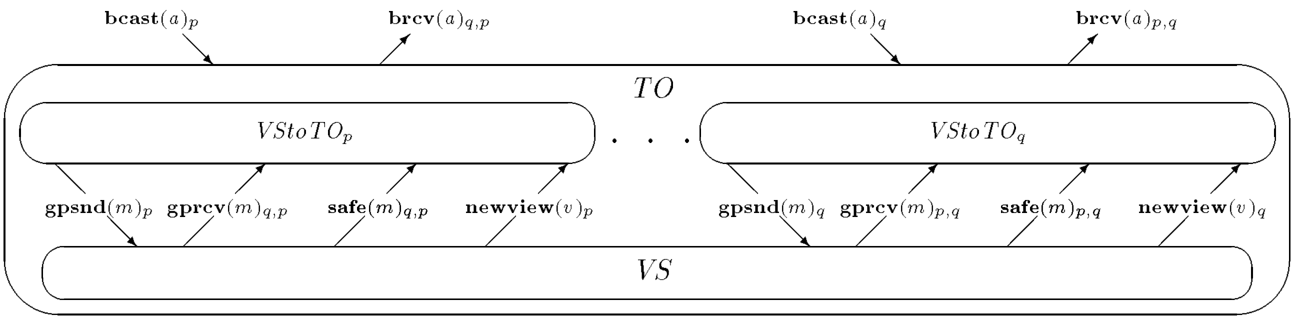

Figure 1: System components and interfaces

VStoTO, can be seen as a more abstract form of both previous ones, separated from the specific use for data replication.

In Sections 6 and 7, we prove 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. The performance and fault-tolerance aspects of the proof involve operational reasoning about timed executions.

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

The full version of this paper can be found at the URL `http://theory.lcs.mit.edu/tds/vsgc.html`.

2 Mathematical Foundations

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 $1 \le i \le length(s)$ 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 $s \le t$, 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 $s \le t$ or $t \le s$. If S is a consistent collection of sequences, we define $lub(S)$ to be the minimum sequence t such that $s \le t$ 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 $i \le length(t)$, $t(i) = f(s(i))$.

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-prop*, 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.

The interface between the 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 of a 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. 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 code is given in Figure 2.

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*

The finite traces of this automaton are exactly the finite prefixes of traces of a totally ordered causal broadcast service, 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.

Now we define the performance/fault-tolerance property *TO-prop*. Its signature *TO-fsig* is the same as the signature of *TO-machine*, 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*.

The intention (though this is formally meaningless at this level of abstraction) is that a *good* processor takes steps with no time delay after they become enabled, a *bad* processor is stopped, and an *ugly* processor 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 the property $TO\text{-}prop(b, d, Q)$ 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.

$TO\text{-}prop(b, d, Q)$**:**
Both of the following hold:

1. β with 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 $\max\{t, (l + l')\} + d$.

We define the specification $TO(b, d, Q)$ to be the pair consisting of *TO-machine* and $TO\text{-}prop(b, d, Q)$. 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 $TO\text{-}prop(b, d, Q)$.

4 View-Synchronous Group Communication

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

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.

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, $p \in S$, 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

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, $m \in M$, $p \in P$, $q \in P$, $v \in \mathit{views}$
  $\mathbf{newview}(v)_p$, $v \in \mathit{views}$, $p \in P$, $p \in v.\mathit{set}$

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

States:

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

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

Transitions:

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

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

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

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

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

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

Figure 3: *VS-machine*

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

Now we define the performance/fault-tolerance property *VS-prop*. Its signature *VS-fsig* is the same as the signature of *VS-machine*, with the addition of failure status actions (as before). We define *VS-prop* as a parameterized property of a timed sequence pair (β, ∞) over external actions of *VS-fsig*. Parameters b and d are nonnegative reals, and Q is a set of processors.

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

1. β with 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 $\max\{t, (l + l')\} + d$.

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

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

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 ⟨label,value⟩ pair in 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

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: $VStoTO_p$

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-sys* 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-sys* to *TO-machine*, established with the help of a series of invariant assertions for *VStoTO-sys*. We add some derived variables to the state of *VStoTO-sys*, 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 four conditions holds:

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 $x = gotstate(p)_q$.

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-sys*. 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-sys*: 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.

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. For example, these invariants demonstrate that *allcontent* is a function from labels to data values, and that information received from a processor p is consistent with p's own knowledge. They show upper and lower bounds on the *highprimary* values. Other invariants assert that information present in the summaries in *allstate* reflects correct summary information for established views (as summarized in the history variables *established* and *buildorder*).

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.

Invariant 1 *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 $\sigma \le buildorder[p, v.id]$. Then for every $x \in allstate$ with $x.high > v.id$, $\sigma \le x.ord$.*

The following invariant says that, once all members of a primary view v agree on a prefix σ of *order*, all summaries with views at least as high as v will also include sequence σ. Its proof uses Invariant 1.

Invariant 2 *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 $\sigma \le buildorder[p, v.id]$. Then for every $x \in allstate$ with $x.high \ge v.id$, $\sigma \le x.ord$.*

Other invariants assert the consistency of the sequences *order* and *confirm* throughout the system. For example:

Invariant 3 *If $x \in allstate$ then the following is true: There exists $v \in created$ such that $v.id \le x.high$, $v.set$ contains a quorum, and for every $q \in v.set$, $established[v.id]_q$ and $x.confirm \le buildorder[q, v]$.*

Invariant 4 *If $x, x' \in allstate$ then*
1. If $x.high \le x'.high$ then $x.confirm \le x'.ord$.
2. Either $x.confirm \le x'.confirm$ or $x'.confirm \le x.confirm$.

Invariant 4 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 $lub_{x\in allstate}(x.confirm)$.

Next, we define the simulation relation f. We define it as a function from reachable states of *VStoTO-sys* to states of *TO-machine*. (We assume an arbitrary default value for unreachable states.) Namely, if x is a reachable state of *VStoTO-sys*, 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 4, which yields the definedness of *allconfirm*.

Lemma 6.1 *Function f is a forward simulation.*

Proof. By induction. The proof amounts to showing that the initial states are related by f, and showing that this relationship is preserved by all steps of the system. In showing that the steps preserve the relationship, some of the invariants are used (for example, Invariant 4). □

Theorem 6.2 *Every trace of VStoTO-sys 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 called $VStoTO'_p$ 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-status* $\neq$ *bad*.
- Add a code fragment for each $\nu(t)$:

```
ν(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'-sys* 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'-sys* satisfies *TO-prop*, for certain values of the parameters:

Theorem 7.1 *Every admissible timed trace of VStoTO'-sys satisfies TO-prop(b+d, d, Q) for every Q that contains a quorum.*

Proof. Let (β, ∞) be any admissible timed trace of *VStoTO'-sys*, and let α be an admissible timed execution of *VStoTO'-sys* 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-prop*: 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.

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-prop* of α. *VStoTO-prop* uses the "conclusion" part of *VS-prop*(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-prop*($b + d, d, Q$).

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. $ltime(\alpha''') \leq d$
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-prop* operationally. In our proof, the execution fragment α''' whose existence is asserted in the conclusion of *VStoTO-prop* 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 the *VStoTO-prop*, it is easy to unwind the definitions of *VS-prop*(b, d, Q) and *TO-prop*(b+d, d, Q) and prove that *VStoTO'-sys* satisfies *TO-prop*($b + d, d, Q$). Namely, suppose as in the hypothesis of *TO-prop*($b + d, d, Q$) that the failure-status actions stabilize in Q. Then by the property *VS-prop*(b, d, Q), within time at most b, the *VS* layer stabilizes to a situation in which there are no view changes, view information is consistent within Q, and messages among processors in Q are delivered (and made safe) within time d.

At this point, the hypothesis of *VStoTO-prop* has been proved; we next apply *VStoTO-prop*, which says that in an additional time at most d, the system stabilizes to a situation where all client-level data values are delivered within time d. This is as needed for *TO-prop*($b + d, d, Q$). □

Theorem 7.1 yields the main result:

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

8 Implementing *VS*

An implementation of *VS* can be constructed from the 3-round membership protocol of [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 which 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 a reasonable amount of 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.

9 Conclusions

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.

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, Dept 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 Int-l 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", TR 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 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 Int-l 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", Tech. 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.