A Dynamic View-Oriented Group Communication Service

Roberto De Prisco* Alan Fekete† Nancy Lynch* Alex Shvartsman‡

January 9, 1998

Abstract

View-oriented group communication services are widely used for fault-tolerant distributed computing. For applications involving coherent data, it is important to know when a process has a view of the current group membership that is *primary*, usually defined as a view containing a majority out of a *static* universe of processes. For high availability in a system where processes can join and leave routinely, some researchers have suggested defining primary views in a *dynamic* way, depending on having enough members in common with recent views.

We present a new formal specification, DVS, for the safety guarantees made by a practical group communication service providing a dynamic notion of primary view. The specification is a simple automaton, with only seven kinds of actions. We demonstrate the value of the DVS specification by showing both how it can be implemented and how it can be used in an application. Both pieces are shown formally, with assertional proofs.

First, we consider an implementation that is a variant of the group membership algorithm of Lotem, Keidar, and Dolev [29]. Our variant integrates communication with the membership service, uses information from the application processes saying when a view has been adequately prepared for computation by the application, and uses a static view-oriented service internally. We prove that this algorithm implements DVS, in the sense of trace inclusion.

Second, we consider an application algorithm that is a variant of an algorithm in [2, 20, 26], modified to use DVS instead of a static view-oriented service. We show that it implements a (non-group-oriented) totally-ordered-broadcast service.

Because of its simplicity, we believe that DVS will be easy to use for other applications.

Paper format: Regular paper

Contact author: Alex Shvartsman, Dept. of Computer Science and Eng., 191 Auditorium Road, U-155, University of Connecticut, Storrs, CT 06269. E-mail: `alex@theory.lcs.mit.edu`.

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

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

‡MIT Laboratory for Computer Science, 545 Technology Square, NE43-371, Cambridge, MA 02139, USA and Dept. of Computer Science and Eng., 191 Auditorium Rd., U-155, University of Connecticut, Storrs, CT 06269.

1 Introduction

View-oriented group communication services have become important as building blocks for practical fault-tolerant distributed systems. Such a service enables application processes located at different nodes of a fault-prone distributed network to operate collectively as a group, using the service to multicast messages to all members of the group. Each such service is based on a *group membership service*, which provides each group member with a *view* of the group; a view includes a list of the processes that are members of the group. Messages sent by a process in one view are delivered only to processes in the membership of that view, and only when they have the same view. Within each view, the service offers guarantees about the order and reliability of message delivery. Examples of view-oriented group communication services are found in Isis [7], Transis [15], Totem [33], Newtop [18], Relacs [4], and Horus [36].

In order to be useful to application programmers, system building blocks should come equipped with simple and precise specifications of their guaranteed behavior. But producing good specifications for view-oriented group communication services is a difficult task, because these services can be quite complicated, and because there are many different such services, with different guarantees about safety, performance, and fault-tolerance. Some examples of specifications for group membership services and view-oriented group communication services appear in [5, 6, 8, 11, 16, 20, 22, 23, 24, 32, 34, 37, 38].

In [20], we presented a new specification, which we called VS, for a view-oriented group communication service. This specification consists of a simple state machine expressing safety requirements, plus a timed trace property expressing conditional performance/fault-tolerance requirements. This specification is easy to read and easy to use; we used it to prove the correctness of a complex totally-ordered-broadcast algorithm based on [26, 2]. Since then, Chockler has used a slightly simplified version of VS to model and verify an adaptive totally-ordered-broadcast algorithm [9], Lesley and Fekete [28] have proved that a version of an algorithm of Cristian and Schmuck [12] implements VS, and Khazan has been using VS to model and verify a load-balancing database application [27].

The VS service produces arbitrary views, with arbitrary membership sets. However, in many applications of VS, especially those with strong data coherence requirements, the application processes perform significant computations only when they have a special type of view called a *primary view*. For example, a replicated database application might only perform a read or write operation within a primary view, in order to ensure that each read receives the result of the last preceding write, in some consistent order of the operations. In this setting, a primary view is typically defined to be one whose membership comprises a *majority* of the complete group of processes, or more generally, a *quorum* in a pre-defined quorum set in which all pairs of quorums intersect. The intersection property guarantees enough information flow between successive primary views to achieve the coherence requirements.

Pre-defined quorum sets can yield efficient implementations in settings with transient process and channel failures. However, they work less well in settings where the underlying system configuration evolves over time, with processes joining and leaving the system. For such a setting, a *dynamic notion of primary* is needed, one that can change to conform with the system configuration. A dynamic notion of primary still needs to maintain some kind of intersection property, in order to guarantee enough information flow between successive primary views to achieve coherence. As a minimum, each primary must contain at least a majority of the processes in the previous primary. Several *dynamic voting schemes* have been developed to define primaries adaptively [13, 17, 25, 29, 35].

In particular, Lotem, Keidar, and Dolev [29] have described an implementation of a group

membership service that yields only primary views, according to a dynamic notion of primary. An interesting feature of their work is that it points out various subtleties of implementing such a membership service in a distributed manner – subtleties involving different opinions by different processes about what is the previous primary view. These difficulties have led to errors in some of the past work on dynamic voting. The algorithm of [29] copes with these subtleties by maintaining information about a collection of primary views that "might be" the previous primary view. The service deals with group membership only, and not with communication. Lotem et al. prove that their protocol satisfies the following condition: any two (primary) views that occur in an execution are linked by a chain of views where for every consecutive pair of views in the chain, there is some process that "knew" it belonged to both views. This correctness condition is formulated as a property of system executions.

In this paper, we present a new formal specification, DVS, for the safety guarantees made by a practical dynamic view-oriented group communication service. This service is inspired by the implementation of Lotem et al., but integrates communication with the group membership service. Our service specification is given by a global automaton instead of a property of system executions.

One important feature of our specification is our careful handling of the interface between the service and the application. In particular, we allow application processes to *register* views; when an application process registers a view, it means that that process is ready to begin ordinary processing using that view. This device allows the process to indicate the completion of any application-dependent pre-processing (typically, an exchange of information) that is needed to prepare for ordinary computation. (For example, processes in a coherent database application may need to exchange information about previous updates in order to bring everyone in the new view up to date. But an application with short-lived data might not need to exchange anything.) The DVS service uses registration information when it creates a new view v, in order to determine which previously-created views must satisfy the intersection property with respect to v. This registration information appears to be important for the correctness of coherent-data applications run over a dynamic view-oriented group communication service.

Another important feature of our specification, compared to that in [29], is that our specification is given as an automaton, which maintains state information about the views and the messages sent in each view. This global state can be used in invariants and abstraction functions, leading to assertional proofs of the correctness of implementations of DVS, and also of applications built over DVS. In contrast, Lotem et al. use a specification given in terms of the whole sequence of events in an execution, and therefore must use operational reasoning about complex sequences of events. Extensive experience with proofs of distributed algorithms suggests that assertional techniques are less error-prone; also they are more amenable to automated checking.

Our DVS specification is a very simple automaton, with only seven kinds of actions. We believe that it is simple enough to be useful for both application designers and communication service implementers. We demonstrate the value of the specification by showing both how it can be implemented and how it can be used in one application. Both pieces are shown formally, with assertional proofs.

First, we consider an implementation that is a variant of the group membership algorithm of Lotem et al.; our variant integrates communication with the membership service, uses registration information from the application processes, and uses a static view-oriented service (VS with a tiny modification) internally. We prove that this algorithm implements DVS, in the sense of trace inclusion. The proof uses an abstraction function and invariant assertions. The key to the proof is an invariant expressing a strong condition about nonempty intersections of views; the proof of this depends on relating a *local* check of *majority* intersection with known views to a *global* check of *nonempty* intersection with existing views.

Second, we consider an application algorithm that is a variant of an algorithm in [26, 2, 20], modified to use DVS instead of a static view-oriented service. The modified algorithm uses the registration capability to tell the DVS service that information has been successfully exchanged at the beginning of a new view. We show that it implements a (non-group-oriented) totally-ordered-broadcast service. The proof uses an abstraction function and invariant assertions.

In Section 2 we overview the mathematical notation for this paper. The specification of the VS service used by DVS is in Section 3. The DVS service is presented in Section 4, and its implementation in Section 5. In Section 6 we use DVS to implement a totally ordered broadcast service.

2 Notation and mathematical foundations

We denote by $\mathcal{P}$ the universe of all processors[1]. We denote by $\mathcal{M}$ the universe of all possible messages. We denote by $\mathcal{G}$ a totally ordered set of identifiers used to distinguish views. Given a set S, the notation $S_\perp$ refers to the set $S \cup \{\perp\}$. We extend the ordering of $\mathcal{G}$ to the set $\mathcal{G}_\perp$ by letting $\perp < g$ for any $g \in \mathcal{G}$.

A view $v = \langle g, P \rangle$ consists of a view identifier g, $g \in \mathcal{G}$ and a set P, $P \in 2^{\mathcal{P}}$, of processors, which are members of the view. Hence the set $\mathcal{V}$ of all views is $\mathcal{V} = \mathcal{G} \times 2^{\mathcal{P}}$. Given a view $v = \langle g, P \rangle$, the notation $v.id$ refers to the view identifier g of view v and the notation $v.set$ refers to the view membership set P of view v. In the initial state of the system there exists a single view $v_0 = \langle g_0, P_0 \rangle$, where g_0 is a distinguished view identifier in $\mathcal{G}$ and P_0, a subset of $\mathcal{P}$, is the group membership.

We write λ for the empty sequence. A sequence of one element x is written as $\langle\langle x \rangle\rangle$. If a is a sequence, $|a|$ denotes the length of a. If a is a sequence and $1 \leq i \leq |a|$ then $a(i)$ denotes the ith element of a. Given a set S, $seqof(S)$ denotes the set consisting of all finite sequences of elements of S. If s and t are sequences, the concatenation of s and t is denoted by $s{+}t$. We also abuse this notation slightly by letting s or t be a single element. We say that sequence s is a *prefix* of sequence t, written as $s \leq t$, provided that there exists u such that $s{+}u = t$. The "head" of a sequence a is $a(1)$. A sequence can be used as a queue: the "append" operation modifies the sequence by concatenating the sequence with a new element and the "remove" operation modifies the sequence by deleting the head of the sequence.

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 R is a binary relation, then we define $dom(R)$, the *domain* of R to be the set, without repetitions, of first elements of the ordered pairs comprising relation R.

The services and algorithms are described using state machine models. Specifically, we use the I/O automaton model of Lynch and Tuttle [31] (without fairness). The model and associated methodology is described in Chapter 8 of [30].

3 The VS specification

In this work we use a modified version of the group communication service defined in [20]. The original service, called VS, assumes that every processor in the universe $\mathcal{P}$ of processors is a member of the initial view. In our setting the initial view is defined arbitrarily, and we modify the specification of VS to reflect this fact. The modified specification is given in Figure 1.

The specific modifications to the service specification in [20] are:

- Initialization of the state variable *created* is changed from $\{v_0\}{=}\{\langle g_0, \mathcal{P} \rangle\}$ to $\{v_0\}{=}\{\langle g_0, P_0 \rangle\}$.
- The type of state variables *current-viewid*$[p]$ is changed from $\mathcal{G}$ to $\mathcal{G}_\perp$; the initialization of *current-viewid*$[p]$ is changed from g_0 for all $p \in \mathcal{P}$ to g_0 for $p \in P_0$ and $\perp$ for $p \notin P_0$.

[1] We use the terms *process* and *processor* interchangeably as any differences are immaterial in our setting.

VS (modified version)

Signature:

Input: VS-GPSND$(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$
Internal: VS-CREATEVIEW(v), $v \in \mathcal{V}$
VS-ORDER(m, p, g), $m \in \mathcal{M}$, $p \in \mathcal{P}$, $g \in \mathcal{G}$
Output: VS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}$, $q, p \in \mathcal{P}$
VS-SAFE$(m)_{q,p}$, $m \in \mathcal{M}$, $q, p \in \mathcal{P}$,
VS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

State:

$created \in 2^{\mathcal{V}}$, init $\{v_0\}$
for each $p \in \mathcal{P}$:
$current\text{-}viewid[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else
for each $g \in \mathcal{G}$:
$queue[g] \in seqof(\mathcal{M} \times \mathcal{P})$, init λ
for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
$pending[p, g] \in seqof(\mathcal{M})$, init λ
$next[p, g] \in \mathbf{N}^{>0}$, init 1
$next\text{-}safe[p, g] \in \mathbf{N}^{>0}$, init 1

Actions:

internal VS-CREATEVIEW(v)
Pre: $\forall w \in created : v.id > w.id$
Eff: $created := created \cup \{v\}$

output VS-NEWVIEW$(v)_p$
Pre: $v \in created$
$v.id > current\text{-}viewid[p]$
Eff: $current\text{-}viewid[p] := v.id$

input VS-GPSND$(m)_p$
Eff: if $current\text{-}viewid[p] \neq \perp$ then
append m to $pending[p, current\text{-}viewid[p]]$

internal VS-ORDER(m, p, g)
Pre: m is head of $pending[p, g]$
Eff: remove head of $pending[p, g]$
append $\langle m, p \rangle$ to $queue[g]$

output VS-GPRCV$(m)_{q,p}$, choose g
Pre: $g = current\text{-}viewid[p]$
$queue[g](next[p, g]) = \langle m, q \rangle$
Eff: $next[p, g] := next[p, g] + 1$

output VS-SAFE$(m)_{q,p}$, choose g, P
Pre: $g = current\text{-}viewid[p]$
$\langle g, P \rangle \in created$
$queue[g](next\text{-}safe[p, g]) = \langle m, q \rangle$
for all $r \in P$:
$next[r, g] > next\text{-}safe[p, g]$
Eff: $next\text{-}safe[p, g] := next\text{-}safe[p, g] + 1$

Figure 1: The VS specification

- The effect of the action VS-GPSND$(m)_p$ is changed so that a message m is appended to the queue $pending[current\text{-}viewid[p]]$ only if $current\text{-}viewid[p]$ is not $\perp$.

These modifications are extremely minor, and the modified service preserves the basic safety properties of the service in [20]:

Monotone views: New views are reported in increasing order of view identifier.

View synchrony: Messages sent in a view are delivered only within that view.

Prefix order: The sequences of messages delivered in a view at any two processors are such that one sequence is a prefix of the other.

In the rest of the paper "VS" refers to the service specified in Figure 1.

4 The DVS specification

We now present a formal specification for a dynamic view-oriented group communication service, called DVS, that provides support for primary components. The DVS specification differs from the VS specification in the following: (*i*) the client is notified only of the views that are primary components, and (*ii*) after the notification of a new view, the client may use the service to gather whatever information it needs and then register with the service the fact that the needed information was obtained. The specification is given in Figure 2.

The DVS service informs its clients of a group status change through a DVS-NEWVIEW$(\langle g, P \rangle)_p$ action (with $p \in P$), which informs processor p that the view identifier g is associated with membership

DVS

Signature:

Input: DVS-GPSND$(m)_p$, $m \in \mathcal{M}_c$, $p \in \mathcal{P}$
DVS-REGISTER$(g)_p$, $g \in \mathcal{G}, p \in \mathcal{P}$

Internal: DVS-CREATEVIEW(v), $v \in \mathcal{V}$
DVS-ORDER(m,p,g), $m \in \mathcal{M}_c$, $p \in \mathcal{P}$, $g \in \mathcal{G}$

Output: DVS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}_c$, $p,q \in \mathcal{P}$
DVS-SAFE$(m)_{q,p}$, $m \in \mathcal{M}_c$, $p,q \in \mathcal{P}$
DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

State:

$created \in 2^{\mathcal{V}}$, init $\{v_0\}$
for each $p \in \mathcal{P}$:
  $current\text{-}viewid[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else
for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
  $pending[p,g] \in seqof(\mathcal{M}_c)$, init λ
  $next[p,g] \in \mathbf{N}^{>0}$, init 1
  $next\text{-}safe[p,g] \in \mathbf{N}^{>0}$, init 1
for each $g \in \mathcal{G}$:
  $queue[g] \in seqof(\mathcal{M}_c \times \mathcal{P})$, init λ
  $registered[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else

Derived variables:
  $active \in \mathcal{G}_\perp$, defined as $active = \max G$ where
    $G = \{v.id | v \in created, v.set \subseteq registered[g]\} \cup \{\perp\}$

Actions:

internal DVS-CREATEVIEW(v)
  Pre: $\forall w \in created : v.id \neq w.id$
    $\forall w \in created$:
      $(\exists x : x.set \subseteq registered[x.id]$ and
        $(w.id < x.id < v.id$ or $v.id < x.id < w.id))$
      or $v.set \cap w.set \neq \{\}$
  Eff: $created := created \cup \{v\}$

output DVS-NEWVIEW$(v)_p$
  Pre: $v \in created$
    $v.id > current\text{-}viewid[p]$
  Eff: $current\text{-}viewid[p] := v.id$

input DVS-REGISTER$(g)_p$
  Eff: $registered[g] := registered[g] \cup \{p\}$

input DVS-GPSND$(m)_p$
  Eff: if $current\text{-}viewid[p] \neq \perp$ then
    append m to $pending[p, current\text{-}viewid[p]]$

internal DVS-ORDER(m,p,g)
  Pre: m is head of $pending[p,g]$
  Eff: remove head of $pending[p,g]$
    append $\langle m,p \rangle$ to $queue[g]$

output DVS-GPRCV$(m)_{q,p}$, choose g
  Pre: $g = current\text{-}viewid[p]$
    $queue[g](next[p,g]) = \langle m,q \rangle$
  Eff: $next[p,g] := next[p,g] + 1$

output DVS-SAFE$(m)_{q,p}$, choose g, P
  Pre: $g = current\text{-}viewid[p]$
    $\langle g, P \rangle \in created$
    $queue[g](next\text{-}safe[p,g]) = \langle m,q \rangle$
    for all $r \in P$:
      $next[r,g] > next\text{-}safe[p,g]$
  Eff: $next\text{-}safe[p,g] := next\text{-}safe[p,g] + 1$

Figure 2: The DVS specification

set P and that the current group of processors connected to p is P. After any finite execution, we define the *current view* at p to be the argument v in the last DVS-NEWVIEW$(v)_p$ event, if any, otherwise it is the pair consisting of the distinguished initial view identifier g_0 and the initial membership P_0 for processors in P_0 and is undefined for other processors. The DVS service guarantees that messages sent by a processor p when the current view of p is some view v are delivered only within that view (i.e., only to processors that are in $v.set$ and whose current view is still v). Moreover each processor receives messages in the same order as any other processor and without gaps in the sequence of received messages (though some processors may receive only a prefix of the sequence of messages received by other processors).

We denote by $\mathcal{M}_c$ the set of messages that clients may use for communication. The set $\mathcal{M}_c$ is a subset of the universe $\mathcal{M}$ of all possible messages. The DVS service allows a processor p to broadcast a message m, $m \in \mathcal{M}_c$, by means of actions of the form DVS-GPSND$(m)_p$. Messages are delivered by means of actions of the form DVS-GPRCV$(m)_{q,p}$, representing the delivery to p of the message m sent by q. The service also provides outputs DVS-SAFE$(m)_{q,p}$ to report to processor p that the earlier message m from q has been delivered to all locations in the current view of q.

Finally the DVS service allows the client to "register" a previously reported view $v = \langle g, P \rangle$ with an action DVS-REGISTER$(g)_p$. With this action, the client at p informs the service that it has obtained whatever information the application needs (for many applications, this will mean that p

has received messages from every other member of view v, reporting their state at the start of g). We assume that the clients register a view only when that view is the latest view the client is notified of by the service. Figure 3 contains the code that formally expresses the above condition.

CLIENT$_p$

Signature:

Input: DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$ Output: DVS-REGISTER$(g)_p$, $g \in \mathcal{G}$

State:

current-viewid $\in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else

Actions:

input DVS-NEWVIEW$(v)_p$
Eff: *current-viewid* $:= v.id$

output DVS-REGISTER$(g)_p$
Pre: $g =$ *current-viewid*
Eff: none

Figure 3: The client

We say that a view v is *totally registered* in a state s of DVS if $v.set \subseteq$ *registered*$[v.id]$. A view becomes totally registered when all clients $p \in v.set$ have executed action DVS-REGISTER$(g)_p$, where $g = v.id$. The second precondition of DVS-CREATEVIEW is the key to our specification; it says that when a view v is created, it must intersect in membership with certain existing views: those with viewid lying in an interval bounded by the totally registered views before and after $v.id$.

5 An implementation of DVS

We now give an algorithm that implements the DVS service, in the sense of trace inclusion. We build the algorithm on top of the VS service and we use ideas from [29]. The overall system is shown in Figure 4. It consists of client CLIENT$_p$ and automaton DVS-CODE$_p$ at each processor p, and the underlying VS service. The automaton DVS-CODE$_p$ is given in Figure 5. The composition of all DVS-CODE$_p$ automata and VS gives the implementation DVS-IMPL. Since we use the well-formedness of the client we consider the system DVS-IMPL-SYS which is the composition of DVS-IMPL and all the CLIENT$_p$ automata with all actions except those at the client boundary hidden. We also define DVS-SYS to be the composition of DVS and all the CLIENT$_p$ automata.

DVS-CODE$_p$ uses special messages internal to the implementation (this means that clients are not allowed to use such messages). These messages are tagged either with *"info"* or *"registered"*. Recall that $\mathcal{M}_c$ is the set of all client messages and $\mathcal{M}$ is the universe of all messages. Here we set $\mathcal{M} = \mathcal{M}_c \cup (\{\text{"}info\text{"}\} \times \mathcal{V} \times 2^{\mathcal{V}}) \cup \{\text{"}registered\text{"}\}$.

The implementation is based on a dynamic voting scheme in which any primary component has a majority intersection with the previous primary component. Whenever a primary component is totally registered, the information about previous primary components can be garbage-collected. However since primary views are reported at different times at each of the members (or even not reported to some of them) and clients may register views at different times, information about primary views after the latest totally registered view must be maintained. Maintaining this infor-

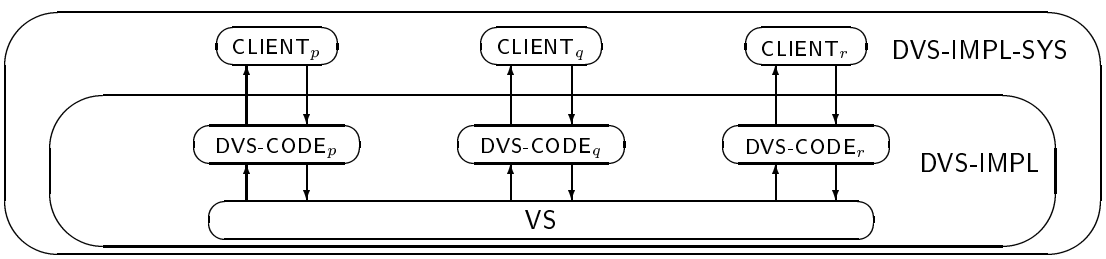

Figure 4: The overall system.

DVS-CODE$_p$

Signature:

Input: DVS-GPSND$(m)_p$, $m \in \mathcal{M}_c$
DVS-REGISTER$(g)_p$, $g \in \mathcal{G}$
VS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$
VS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}$, $q \in \mathcal{P}$
VS-SAFE$(m)_{q,p}$, $m \in \mathcal{M}$, $q \in \mathcal{P}$

Internal: DVS-GARBAGE-COLLECT$(v)_p$, $v \in \mathcal{V}$
Output: VS-GPSND$(m)_p$, $m \in \mathcal{M}$
DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$
DVS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}_c$, $q \in \mathcal{P}$
DVS-SAFE$(m)_{q,p}$, $m \in \mathcal{M}_c$, $q \in \mathcal{P}$

State:

$cur \in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else
$client\text{-}cur \in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else
$act \in \mathcal{V}$, init v_0
$amb \in 2^{\mathcal{V}}$, init $\{\}$
$attempted \in 2^{\mathcal{V}}$, init $\{v_0\}$ if $p \in P_0$, $\{\}$ else
for each $g \in \mathcal{G}$
  $msgs\text{-}to\text{-}vs[g] \in seqof(\mathcal{M})$, init λ
  $info\text{-}sent[g] \in (\mathcal{V} \times 2^{\mathcal{V}})_\perp$, init $\perp$
  $reg[g]$ a bool, init `true` if $p \in P_0$ and $g = g_0$, `false` else
for each $g \in \mathcal{G}$
  $msgs\text{-}from\text{-}vs[g] \in seqof(\mathcal{M}_c \times P)$, init λ
  $safe\text{-}from\text{-}vs[g]] \in seqof(\mathcal{M}_c \times P)$, init λ
for each $g \in \mathcal{G}$, $q \in \mathcal{P}$
  $info\text{-}rcvd[q,g] \in (\mathcal{V} \times 2^{\mathcal{V}})_\perp$, init $\perp$
  $rcvd\text{-}rgst[q,g]$ a bool, init `false`

Derived variables
  $use \in 2^{\mathcal{V}}$, defined as $use = \{act\} \cup amb$

Actions:

input VS-NEWVIEW$(v)_p$
  Eff: $cur := v$
    append $\langle$ *"info"*, act, $amb\rangle$ to $msgs\text{-}to\text{-}vs[cur.id]$
    $info\text{-}sent[cur.id] = \langle act, amb\rangle$

input VS-GPRCV$(\langle$ *"info"*$, v, V\rangle)_{q,p}$
  Eff: $info\text{-}rcvd[cur.id, q] := \langle v, V\rangle$
    if $v.id > act.id$ then $act := v$
    $amb := \{w \in amb \cup V | w.id > act.id\}$

input VS-SAFE$(\langle$ *"info"*$, v, V\rangle)_{q,p}$
  Eff: none

output DVS-NEWVIEW$(v)_p$
  Pre: $v = cur$
    $v.id > client\text{-}cur.id$
    $\forall q \in v.set, q \neq p : info\text{-}rcvd[q, v.id] \neq \perp$
    $\forall w \in use : |v.set \cap w.set| > |w.set|/2$
  Eff: $amb := amb \cup \{v\}$
    $attempted := attempted \cup \{v\}$
    $client\text{-}cur := v$

input DVS-REGISTER$(g)_p$
  Eff: $reg[g] :=$ `true`
    append $\langle$ *"registered"*$\rangle$ to $msgs\text{-}to\text{-}vs[cur.id]$

input VS-GPRCV$(\langle$ *"registered"*$\rangle)_{q,p}$
  Eff: $rcvd\text{-}rgst[cur.id, q] :=$ `true`

input VS-SAFE$(\langle$ *"registered"*$\rangle)_{q,p}$
  Eff: none

internal DVS-GARBAGE-COLLECT$(v)_p$
  Pre: $\forall q \in v.set : rcvd\text{-}rgst[v, q] =$ `true`
    $v.id > act.id$
  Eff: $act := v$
    $amb := \{w \in amb | w.id > act.id\}$

input DVS-GPSND$(m)_p$
  Eff: append m to $msgs\text{-}to\text{-}vs[client\text{-}cur.id]$

output VS-GPSND$(m)_p$
  Pre: m is head of $msgs\text{-}to\text{-}vs[cur.id]$
  Eff: remove head of $msgs\text{-}to\text{-}vs[cur.id]$

input VS-GPRCV$(m)_{q,p}$, where $m \in \mathcal{M}_c$
  Eff: append $\langle m, q\rangle$ to $msgs\text{-}from\text{-}vs[cur.id]$

output DVS-GPRCV$(m)_{q,p}$
  Pre: $\langle m, q\rangle$ is head of $msgs\text{-}from\text{-}vs[client\text{-}cur.id]$
  Eff: remove head of $msgs\text{-}from\text{-}vs[client\text{-}cur.id]$

input VS-SAFE$(m)_{q,p}$, where $m \in \mathcal{M}_c$
  Eff: append $\langle m, q\rangle$ to $safe\text{-}from\text{-}vs[cur.id]$

output DVS-SAFE$(m)_p$
  Pre: $\langle m, q\rangle$ is head of $safe\text{-}from\text{-}vs[client\text{-}cur.id]$
  Eff: remove head of $safe\text{-}from\text{-}vs[client\text{-}cur.id]$

Figure 5: The DVS implementation.

mation, as pointed out in [29], is crucial to preserving consistency. Thus each processor p has an "active" view, act_p which is the latest view it knows is totally registered. Information about views previous to act_p is deleted. Views after act_p are kept in amb_p. These views are called "ambiguous" since processor p does not know whether they are totally registered or not. For simplicity of notation we consider the set $use_p = \{act_p\} \cup amb_p$. To attempt a new primary component, the primary component must have a majority intersection with each of the views in the use_p set because any of these might not have a totally registered view between it and the new view.

The underlying VS provides information about a new view by means of action VS-NEWVIEW. Processors that receive the new view send out some information (*"info"*) messages about past

computation consisting of the active view, act_p, and the ambiguous views, amb_p. Members of the new view expect to receive *"info"* messages from all other members. When all such messages are received by some processor p, the processor checks that the current view has majority intersection with the latest view known to be totally registered, and all the subsequent views that have been reported to the client. If the majority intersection holds for all these views, processor p "attempts" the view: the client is notified of the new view by means of action DVS-NEWVIEW. Then, the clients can use the communication service to exchange information within the same view. When a client has obtained enough information, it "registers" the view by means of action DVS-REGISTER. When a view is registered by the client of processor p, processor p sends a *"registered"* message to the other members. Action DVS-GARBAGE-COLLECT is enabled when *"registered"* messages are received from all members. A processor executing this action may update its knowledge of totally registered views, and perform garbage collection by discarding attempted views with identifiers smaller than that of the latest view known to be totally registered.

DVS-CODE uses VS to send messages. Hence whenever a client executes a DVS-GPSND the message to be sent is passed to VS by means of action VS-GPSND. Similarly whenever VS delivers a message with action VS-GPRCV (or a safe notification with VS-SAFE) the message is passed to the client with DVS-GPRCV (or the safe notification with DVS-SAFE). Since the client is notified about a new view only when the view is a primary, it is possible that the client has an old view with respect to the latest one provided by VS. We make sure that the view of the underlying VS and the view of the client are the same by keeping track of the client's current view so that the incoming and outgoing messages are placed into appropriate buffers.

We now outline the proof that DVS-IMPL-SYS implements DVS-SYS. We first state the main theorem and then describe the main facts used in the proof (in the full paper).

Theorem 5.1 *Every trace of* DVS-IMPL-SYS *is a trace of* DVS-SYS.

We prove that DVS-IMPL-SYS implements DVS-SYS by defining a function $\mathcal{F}$ that maps the states of DVS-IMPL-SYS to the states of DVS-SYS and by proving that this function is an *abstraction function*. An abstraction function $\mathcal{F}$ is required to satisfy the following two conditions: (1) If s is any initial state of DVS-IMPL-SYS, then $\mathcal{F}(s)$ is an initial state of DVS-SYS, and (2) If s and $\mathcal{F}(s)$ are reachable states of DVS-IMPL and DVS-SYS respectively, and (s, π, s') is a step of DVS-IMPL-SYS, then there is a sequence of steps of DVS-SYS from $\mathcal{F}(s)$ to $\mathcal{F}(s')$, having the same trace.

The state of DVS-IMPL-SYS includes the state of the VS service. Given a state s of DVS-IMPL, we use the notation $s.\text{VS}.x$ to refer to component x of the state of the VS service. DVS-IMPL-SYS uses VS to send both the messages internal to the implementation (*"info"* and *"registered"* message) and the messages sent by the client. Our abstraction function needs to discard the internal messages when mapping the states of DVS-IMPL whose queues contain both types of messages to the states of DVS-SYS. We define the *purge* function which takes as argument a queue of messages q and returns a new queue q' obtained by deleting from q any *"info"* and *"registered"* messages. We also define the function *purgesize* that, given a queue, returns the number of internal messages in it.

Figure 6 shows the abstraction function $\mathcal{F}$.

For DVS we have the notion of totally registered view. In DVS-IMPL-SYS we say that a view v is *totally registered* in state s, if $s.reg[v.id]_p = \texttt{true}$ for all $p \in v.set$. Notice that v is totally registered in state s of DVS-IMPL-SYS if and only if it is totally registered in the state of DVS that appears in state $\mathcal{F}(s)$ of DVS-SYS. $\mathcal{TOTREG}$ is a derived variable of DVS-IMPL-SYS that denotes the set of totally registered views. Also, we say that a view v is *attempted* in state s of DVS-IMPL-SYS if $v \in s.attempted_p$ for some p. $\mathcal{ATT}$ is a derived variable of DVS-IMPL-SYS that denotes the set of attempted views.

The abstraction function $\mathcal{F}$.
Let s be a state of DVS-IMPL-SYS. The state $t = \mathcal{F}(s)$ of DVS-SYS is the following.

- The state components of CLIENT correspond
- $t.created = \cup_{p \in \mathcal{P}} s.attempted_p$
- for each $p \in \mathcal{P}$, $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$
- for each $g \in \mathcal{G}$, $t.registered[g] = \{p | s.reg[g]_p\}$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$, $t.pending[p,g] = purge(s.msgs\text{-}to\text{-}vs[g]_p) + purge(s.\text{VS}.pending[p,g])$
- for each $g \in \mathcal{G}$, $t.queue[g] = purge(s.\text{VS}.queue[g])$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$,
  $t.next[p,g] = s.\text{VS}.next[p,g] - purgesize(s.\text{VS}.queue[g]) - |s.msgs\text{-}from\text{-}vs[g]_p|$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$,
  $t.next\text{-}safe[p,g] = s.\text{VS}.next\text{-}safe[p,g] - purgesize(s.\text{VS}.queue[g]) - |s.safe\text{-}from\text{-}vs[g]_p|$

Figure 6: The abstraction function.

We now state the main invariant. This invariant expresses a global property satisfied by attempted views. Informally this invariant says that any attempted view v intersects the latest totally registered view among those with view identifer less than $v.id$, in a majority of the processors of that totally registered view.

Invariant 5.2 *In any reachable state of* DVS-IMPL-SYS*, consider a view $v \in \mathcal{A}tt$.*
If there exists a view $t \in \mathcal{T}ot\mathcal{R}eg$ such that $t.id = \max\{x.id | x \in \mathcal{T}ot\mathcal{R}eg, x.id < v.id\}$,
then we have that $|v.set \cap t.set| > |t.set|/2$.

The following invariant, a corollary to Invariant 5.2, is instrumental in proving that $\mathcal{F}$ is an abstraction function from DVS-IMPL-SYS to DVS-SYS.

Invariant 5.3 *In any reachable state of* DVS-IMPL-SYS*, the following is true:*
If $v, w \in \mathcal{A}tt$, $v.id < w.id$, and $\nexists x \in \mathcal{T}ot\mathcal{R}eg$ with $v.id < x.id < w.id$, then $v.set \cap w.set \neq \{\}$.

The next lemma shows that $\mathcal{F}$ is an abstraction function.

Lemma 5.4 *(i) If s is an initial state of* DVS-IMPL-SYS *then $\mathcal{F}(s)$ is an initial state of* DVS-SYS*; (ii) Let s be a reachable state of* DVS-IMPL *and $t = \mathcal{F}(s)$ be the corresponding reachable state of* DVS*. For every possible step (s, π, s') there is a sequence σ of steps of the specification that goes from $t = \mathcal{F}(s)$ to $t' = \mathcal{F}(s')$ such that $trace(\sigma) = trace(\pi)$.*

In proving part (ii) of this lemma we consider all possible actions π. The most involved case is the one dealing with the action DVS-NEWVIEW$(v)_p$, which is shown with the help of Invariant 5.3. Having shown that $\mathcal{F}$ is an abstraction function, we infer Theorem 5.1.

6 Using the specification

In this section we show how to use DVS to implement a totally ordered broadcast service. We use the same formal specification provided in [20], called TO. The specification is shown in Figure 7. We provide an implementation of the TO service using DVS as a building block.

This implementation is similar to the TO implementation provided in [20] but exploits the extra information provided by the DVS service. The TO implementation of [20] is built on top of the VS service which does not provide information about primary views. The TO implementation [20] relies on primary views to establish the total order, thus it needs to determine whether or not the current view is primary, which is accomplished by a simple local test. Our implementation is somewhat

TO

Signature:

Input: $\text{BCAST}(a)_p$, $a \in \mathcal{A}$, $p \in \mathcal{P}$
Internal: $\text{TO-ORDER}(a, p)$, $a \in \mathcal{A}$, $p \in \mathcal{P}$
Output: $\text{BRCV}(a)_{p,q}$, $a \in \mathcal{A}$, $p, q \in \mathcal{P}$

State:

$queue \in seqof(\mathcal{A} \times \mathcal{P})$, init λ
for each $p \in \mathcal{P}$: $pending[p] \in seqof(\mathcal{A})$, init λ
$next[p] \in \mathbf{N}^{>0}$, init 1

Actions:

input $\text{BCAST}(a)_p$
Eff: append a to $pending[p]$

internal $\text{TO-ORDER}(a, p)$
Pre: a is head of $pending[p]$
Eff: remove head of $pending[p]$
append $\langle a, p \rangle$ to $queue$

output $\text{BRCV}(a)_{p,q}$
Pre: $queue(next[q]) = \langle a, p \rangle$
Eff: $next[q] := next[q] + 1$

Figure 7: The TO specification.

simpler because it does not require this test (DVS reports only primary views) and because it does not need special processing while the view is not primary. Except for these differences, the TO implementation is as in [20]. The overall implementation, called TO-IMPL, is obtained by composing an automaton TO-CODE_p for each $p \in \mathcal{P}$ with the DVS specification. The complete code for automaton TO-CODE_p is given in Figure 8 in the Appendix.

We prove that every trace of TO-IMPL is also a trace of TO by defining a function $\mathcal{F}_{TO}$ from the reachable states of TO-IMPL to the states of TO and proving that $\mathcal{F}_{TO}$ is an abstraction function. Figure 9 in the Appendix shows the abstraction function $\mathcal{F}_{TO}$.

The proof that $\mathcal{F}_{TO}$ is an abstraction function from TO-IMPL to TO follows the outline of the one in [21]. However, the main invariant requires a different proof. That invariant is as follows.

Lemma 6.1 *The following is true in all reachable states of* TO-IMPL*:*
Suppose that $v \in created$, $\sigma \in seqof(\mathcal{L})$, 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 $x.high > v.id$, $\sigma \leq x.ord$.

This invariant has a more subtle proof then the one given in Lemma 6.17 of [21]. That proof uses the strong intersection property among primary view membership (in the implementation of [21] each primary view intersects each other primary view). The proof in [21] does not work in the setting of DVS because DVS guarantees a weaker intersection property (each primary view intersects only the primary views in between the preceding and the following totally registered primary views). The new proof also uses the fact about DVS that once a view is attempted at all processes in its membership set, no views with lower identifiers can become established.

7 Discussion

In this work we concentrated on safety properties. In future work we plan to consider performance and fault-tolerance as well.

Acknowledgments: We thank Ken Birman who, after reading our work on VS, urged us to consider the more complicated issues of dynamic views. We also thank Idit Keidar for discussions about our DVS specification and our algorithm models and proofs.

References

[1] *Communications of the ACM*, special section on group communications, vol. 39, no. 4, 1996.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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*, pp 72–86, 1995.

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

[8] T.D. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost, "On the Impossibility of Group Membership", in *Proc. of* 15^{th} *Annual ACM Symp. on Principles of Distributed Comp.*, pp. 322-330, 1996.

[9] G.V. Chockler, "An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions", manuscript, Institute of Computer Science, The Hebrew University of Jerusalem, August, 1997.

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

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

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

[13] D. Davcev and W. Buckhard, "Consistency and recovery control for replicated files", in *ACM Symp. on Operating Systems Principles*, n.10, pp. 87–96, 1985.

[14] S.E. Deering and D.R. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs", *ACM Trans. on Computer Systems*, vol. 8, no. 2, 1990.

[15] 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.

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

[17] A. El Abbadi and S. Dani, "A dynamic accessibility protocol for replicated databases", *Data and knowledge engineering*, n.6, pp. 319–332, 1991.

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

[19] 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.

[20] A. Fekete, N. Lynch and A. Shvartsman "Specifying and using a partitionable group communication service", in *Proc. of the* 16^{th} *annual ACM Symposium on Principles of Distributed Computing*, Santa Barbara, CA, August 1997, pp. 53–62.

[21] A. Fekete, N. Lynch and A. Shvartsman "Specifying and using a partitionable group communication service", extended version of [20], available at `http://theory.lcs.mit.edu/tds`

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

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

[24] F. Jahanian, S. Fakhouri and R. Rajkumar, "Processor Group Membership Protocols: Specification, Design and Implementation", in *Proc. of* 12^{th} *IEEE Symp. on Reliable Distrib. Systems* pp 2–11, 1993.

[25] S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database", *ACM Trans. Database Systems*, n.15(2), pp. 230–280, 1990.

[26] I. Keidar and D. Dolev, "Efficient Message Ordering in Dynamic Networks", in *Proc. of* 15^{th} *Annual ACM Symp. on Principles of Distributed Computing*, pp. 68-76, 1996.

[27] R. Khazan, "Group Communication as a base for a Load-Balancing, Replicated Data Service", M.S. thesis, to appear, Dept. of Elec. Eng. and Computer Science, Massachusetts Institute of Technology.

[28] N. Lesley and A. Fekete, "Providing virtual synchrony for group communication services", Preprint 1997.

[29] E. Lotem, I Keidar and D. Dolev, "Dynamic voting for consistent primary components", in *Proc. of the* 16^{th} *Annual ACM Symposium on Principles of Distributed Computing*, Santa Barbara, CA, August 1997, pp. 63-71.

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

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

[32] 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.

[33] 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.

[34] G. Neiger, "A New Look at Membership Services", in *Proc. of* 15^{th} *Annual ACM Symp. on Principles of Distributed Computing*, pp. 331-340, 1996.

[35] J. Paris and D. Long, "Efficient dynamic voting algorithms", *Proc. of* 13^{th} *International Conference on Very Large Data Base*, pp. 268–275, 1988.

[36] 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.

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

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

Appendix

In the appendix we give the complete code for TO-CODE$_p$ and define an abstraction function from the states of TO-impl to the states of TO.

The code for TO-CODE$_p$ is in Figure 8. The code uses additional definitions as follows. Let $\mathcal{L}$ be a set of labels defined as $\mathcal{L} = \mathcal{G} \times \mathbf{N}^{>0} \times \mathcal{P}$. Given a label $l = \langle g, n, p \rangle$ we denote with $l.id$ the view identifier g, with $l.seqno$ the integer n and with $l.origin$ the processor p. We denote by $\mathcal{A}$ the set of messages that can be sent by the clients of the TO service. We refer to an element $a \in \mathcal{A}$ as a datum, to distinguish the clients' messages from the messages used by the implementation. A label is used as unique identifier for a datum sent by the client of processor p.

We define the set $\mathcal{C}$ as $\mathcal{C} = \mathcal{L} \times \mathcal{A}$. An element $\langle l, a \rangle$ of this set represents the association of label l with datum a. A *content* $C \in 2^{\mathcal{C}}$ can be seen as a partial function from $\mathcal{L}$ to $\mathcal{A}$. Each datum is given a unique label and contents are used to store the association between labels and data.

We define the set $\mathcal{S}$ of *summaries* by $\mathcal{S} = 2^{\mathcal{C}} \times seqof(\mathcal{L}) \times \mathbf{N}^{>0} \times \mathcal{G}$. Given a summary $x = (C, L, n, g)$ we denote with $x.con$ the content C, with $x.ord$ the sequence L of labels, with

TO-CODE$_p$

Signature:

Input: BCAST$(a)_p$, $a \in \mathcal{A}$
DVS-GPRCV$(m)_{q,p}$, $q \in \mathcal{P}$, $m \in \mathcal{C} \cup \mathcal{S}$
DVS-SAFE$(m)_{q,p}$, $q \in \mathcal{P}$, $m \in \mathcal{C} \cup \mathcal{S}$
DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$

Output: DVS-REGISTER$(g)_p$
DVS-GPSND$(m)_p$, $m \in \mathcal{C} \cup \mathcal{S}$
BRCV$(a)_{q,p}$, $a \in \mathcal{A}$, $q \in \mathcal{P}$

Internal: CONFIRM$_p$

State:

$current \in \mathcal{V} \cup \{\perp\}$, init v_0 if $p \in P_0$, $\perp$ else
$status \in \{normal, send, collect\}$, init $normal$
$content \subseteq \mathcal{L} \times \mathcal{A}$, init $\{\}$
$nextseqno \in \mathbf{N}^{>0}$, init 1
$buffer \in seqof(\mathcal{L})$, init λ
$safe\text{-}labels \in 2^{\mathcal{L}}$, init $\{\}$
$order \in seqof(\mathcal{L})$, init λ
$nextconfirm \in \mathbf{N}^{>0}$, init 1

$nextreport \in \mathbf{N}^{>0}$, init 1
$highprimary \in G$, init g_0
$gotstate$, a partial function from $\mathcal{P}$ to $\mathcal{S}$, init $\{\}$
$safe\text{-}exch \subseteq \mathcal{P}$, init $\{\}$,
for each $g \in \mathcal{G}$, $established[g]$, a bool, init `false`
$registered \subseteq \mathcal{G}$, init $\{g_0\}$ if $p \in P_0$, $\{\}$ else
$delay \in seqof(\mathcal{A})$, init λ.

Actions:

input BCAST$(a)_p$
Eff: append a to $delay$

internal LABEL$(a)_p$
Pre: a is head of $delay$
$current \neq \perp$
Eff: let m be $\langle current.id, nextseqno, p\rangle$
$content := content \cup \{\langle m, a\rangle\}$
append m to $buffer$
$nextseqno := nextseqno + 1$
delete head of $delay$

output DVS-GPSND$(\langle l, a\rangle)_p$
Pre: $status = normal$
l is head of $buffer$
$\langle l, a\rangle \in content$
Eff: delete head of $buffer$

input DVS-GPRCV$(\langle l, a\rangle)_{q,p}$
Eff: $content := content \cup \{\langle l, a\rangle\}$
$order := order + \langle\langle l\rangle\rangle$

input DVS-SAFE$(\langle l, a\rangle)_{q,p}$
Eff: $safe\text{-}labels := safe\text{-}labels \cup \{l\}$

internal CONFIRM$_p$
Pre: $order(nextconfirm) \in safe\text{-}labels$
Eff: $nextconfirm := nextconfirm + 1$

output BRCV$(a)_{q,p}$
Pre: $nextreport < nextconfirm$
$\langle order(nextreport), a\rangle \in content$
$q = order(nextreport).origin$
Eff: $nextreport := nextreport + 1$

input DVS-NEWVIEW$(v)_p$
Eff: $current := v$
$nextseqno := 1$
$buffer := \lambda$
$gotstate := \{\}$
$safe\text{-}exch := \{\}$
$safe\text{-}labels := \{\}$
$status := send$

output DVS-GPSND$(x)_p$
Pre: $status = send$
$x = \langle content, order, nextconfirm, highprimary\rangle$
Eff: $status := collect$

input DVS-GPRCV$(x)_{q,p}$
Eff: $content := content \cup x.con$
$gotstate := gotstate \oplus \langle q, x\rangle$
if $(dom(gotstate) = current.set) \wedge (status = collect)$ then
$nextconfirm := maxnextconfirm(gotstate)$
$order := fullorder(gotstate)$
$highprimary := current.id$
$status := normal$
$established[current.id] := true$

output DVS-REGISTER$(g)_p$
Pre: $established[g]$
$g \notin registered$
Eff: $registered := registered \cup \{g\}$

input DVS-SAFE$(x)_{q,p}$
Eff: $safe\text{-}exch := safe\text{-}exch \cup \{q\}$
if $safe\text{-}exch = current.set$ then
$safe\text{-}labels := safe\text{-}labels \cup$
$range(fullorder(gotstate))$

Figure 8: TO-CODE$_p$

$x.next$ the integer n and with $x.high$ the view identifier g. A summary x_p of processor p contains some crucial information about the status of that processor. This information consists of (i) the content function $x.con$ of processor p, (ii) the order, represented as a sequence of labels $x.ord$, as known by p, (iii) an integer $x.next$ indicating the last element of the sequence $x.ord$ that can be reported to the client (this means that the order up to that element has been fully established, i.e., is consistent over all the processors, and will not change anymore), (iv) a view identifier $x.high$

indicating the highest view identifier of a (primary) view in which the processor has changed its own knowledge about the total order.

Given $x \in \mathcal{S}$, $x.confirm$ is the prefix of $x.ord$ such that $length(x.confirm) = min(x.next - 1, length(x.ord))$.

Let Y be a partial function from processor ids to summaries. We define

$knowncontent(Y) = \cup_{q \in dom(Y)} Y(q).con$,

$maxprimary(Y) = \max_{q \in dom(Y)} \{Y(q).high\}$ and

$maxnextconfirm(Y) = \max_{q \in dom(Y)} Y(q).next$.

Then we define the set of "representatives" as

$reps(Y) = \{q \in dom(Y) : Y(q).high = maxprimary\}$

and $chosenrep(Y)$ denotes some element in $reps(Y)$.

Finally we define $shortorder(Y) = Y(chosenrep(Y)).ord$ and $fullorder(Y)$ as $shortorder(Y)$ followed by the remaining elements of $dom(knowncontent(Y))$, in label order.

We now define the abstraction function $\mathcal{F}_{TO}$. This function is very similar to the one given in [20]. In order to avoid a complete restatement of the definitions of types, variables and basic ideas, we refer the reader to [20].

The abstraction function $\mathcal{F}_{TO}$.

Let s be a state of TO-IMPL. The state $t = \mathcal{F}_{TO}(s)$ of TO is the following.

1. $t.queue = applyall(\langle x.allcontent, origin \rangle, x.allconfirm)$,
   where the selector $origin$ is regarded as a function from labels to processors.
2. $t.next[p] = x.next\text{-}report_p$.
3. $t.pending[p] = applyall(x.allcontent, s) \cdot x.delay_p$ 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.

Figure 9: The abstraction function.