A Dynamic View-Oriented Group Communication Service*

Roberto De Prisco[†] Alan Fekete[‡] Nancy Lynch[†] Alex Shvartsman[§]

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 *primary* view of the current group membership, 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 *dynamically*, depending on having enough members in common with recent views.

We present a new formal automaton specification, DVS, for the safety guarantees made by a practical group communication service providing a dynamic notion of primary view. We demonstrate the value of DVS by showing both how it can be implemented and how it can be used in an application. First, we present a distributed algorithm based on a group membership algorithm of Lotem, Keidar and Dolev; our version integrates communication with the membership service, uses information from the application processes saying when a view has been prepared for computation by the application, and uses a static view-oriented service internally. We prove that this algorithm implements DVS. Second, we present an application algorithm that is a variant of an algorithm of Amir, Dolev, Keidar, Melliar-Smith and Moser, modified to use DVS instead of a static service. We prove that it implements a (non-group-oriented) totally-ordered-broadcast service.

1 Introduction

View-oriented group communication services have become important as building blocks for 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 [4], Transis [8], Totem [22], Newtop [11], Relacs [2], and Horus [24].

For maximum usefulness, system building blocks should have simple and precise specifications of their guaranteed behavior. Producing good specifications for view-oriented group communication services is difficult, because these services can be complicated, and because different such services provide different guarantees. Examples of specifications for group membership services and view-oriented group communication services appear in [3, 5, 6, 9, 14, 15, 21, 25, 26]. In [12], we presented a specification, VS, for a view-oriented group communication service. This specification consists of a state machine expressing safety requirements, plus a timed trace property expressing conditional performance and fault-tolerance requirements. We used this specification as the basis for proving the correctness of a complex totally-ordered-broadcast algorithm based on [17, 1].

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 universe of processes, or more generally, a *quorum* in a predefined quorum set in which all pairs of quorums intersect. The intersection property permits information flow from any previous primary to a newly formed one.

Pre-defined quorum sets can yield efficient implementations in settings where the system configuration is relatively static. However, they work less well in settings where the 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 permit enough information flow between successive primary views to achieve coherence. For example, each primary view might have to contain at least a majority of the processes in the previous primary view. Several *dynamic voting schemes* have been developed to define primaries adaptively [7, 10, 16, 18, 23].

In particular, Lotem, Keidar, and Dolev [18] 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 mem-

*This research was supported by the following contracts: ARPA F19628-95-C-0118, AFOSR F49620-97-1-0337, NSF 9225124-CCR, and by a grant from the GTE Laboratories.

[†]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, USA.

bership 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 [18] 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 on system executions: 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 "knows" it belongs to both views.

In this paper, we present a new formal automaton 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.

We demonstrate the value of our DVS specification by showing both how it can be implemented and how it can be used in an application. 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 saying when a view has been prepared for computation by the application, and uses a static view-oriented service (a version of VS) internally. We prove that this algorithm implements DVS, in the sense of trace inclusion. The proof uses a (single-valued) simulation relation 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 [17, 1, 12], 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. This proof also uses a simulation relation and invariant assertions.

We have designed our DVS specification to express the guarantees that are useful in verifying correctness of applications that use the service. Among previous work, two different sorts of specifications for a primary group service are notable. Work by Ricciardi and others [26] is expressed in temporal logic on consistent cuts; the idea of their specification is that on any cut, there are no disjoint sets of processes such that each set is collectively aware of no members outside that set. Lotem et al. [18] use a property of an execution, which was previously defined by Cristian [6] for majority groups and that links any two (primary) views by a chain of views where every consecutive pair of views includes a process that "knows" it belongs to both views. As far as we know, these previous specifications have not been used to verify any applications running above them.

An important feature of our specification is our careful handling of the interface between the service and the application. When a new view starts, applications generally require some initial pre-processing to prepare for ordinary computation. For example, applications involving coherent data need to collect knowledge of changes from previous views, before allowing further activity. Our specification does not assume that information flows from one view to another merely because some process joins both. Instead, we treat the exchange of information as application-specific: We ask each application process to indicate, with a REGISTER event, when it has received all the needed information from other members of the new view v. When all members have registered v, the application has gathered all information it needs from previous views, and the service no longer needs to ensure intersection in membership between views before v and any subsequent ones that are formed. In contrast, in a specification based on a chain of views with common membership of successive pairs, one must have the application-level state exchange piggybacked on messages within the group management layer.

Our specification also omits some features of existing dynamic primary view management algorithms. For example, Isis [4] guarantees that processes that move together from one view to the next receive exactly the same messages in the first view. These properties are not needed to verify applications such as the one giving a totally-ordered broadcast. Of course there may be other applications that do require these stronger properties.

2 Mathematical foundations

We write λ for the empty sequence. If a is a sequence then $|a|$ denotes the length of a. If a is a sequence and $1 \leq i \leq j \leq |a|$ then $a(i)$ denotes the ith element of a and $a(i..j)$ denotes the subsequence $a(i), a(i+1), ..., a(j)$ of a. The *head* of a nonempty sequence a is $a(1)$. A sequence can be used as a queue: the *append* operation modifies the sequence by concatenating it with a new element and the *remove* operation modifies the sequence by deleting its head.

If a and b are sequences, a finite, then $a{+}b$ denotes the concatenation of a and b. We sometimes abuse this notation by letting a or b be a single element. We say that sequence a is a *prefix* of sequence b, written $a \leq b$, provided that there exists c such that $a{+}c = b$. A collection A of sequences is *consistent* provided that $a \leq b$ or $b \leq a$ for all $a, b \in A$. If A is a consistent collection of sequences, we define $lub(A)$ to be the minimum sequence b such that $a \leq b$ for all $a \in A$.

If S is a set, then $seqof(S)$ denotes the set of all finite sequences of elements of S. If $a \in seqof(S)$ and f is a partial function from S to T whose domain includes the set of all elements of S appearing in a, then $applytoall(f, a)$ denotes the sequence b such that $length(b) = length(a)$ and, for $i \leq length(b)$, $b(i) = f(a(i))$.

If S is a set, the notation $S_\perp$ refers to the set $S \cup \{\perp\}$. 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. If f is a partial function from S to T, and $\langle s,t\rangle \in S \times T$, then $f \oplus \langle s,t\rangle$ is defined to be the partial function that is identical to f except that $f(s) = t$.

$\mathcal{P}$ denotes the universe of all processors,[1] and $\mathcal{M}$ the universe of all possible messages. $\mathcal{G}$ is a totally ordered set of identifiers used to distinguish views, with a distinguished least element g_0. A *view* $v = \langle g, P\rangle$ consists of a view identifier $g \in \mathcal{G}$ and a nonempty membership set $P \subseteq \mathcal{P}$; we write $v.id$ and $v.set$ to denote the view identifier and membership set components of v, respectively. $\mathcal{V}$ denotes the set of all views, and $v_0 = \langle g_0, P_0\rangle$ is a distinguished *initial view*.

We describe our services and algorithms using the I/O automaton model of Lynch and Tuttle [20] (without fairness). The model and its proof methods are described in Chapter 8

[1] We use "processor" and "process" interchangeably.

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$(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$
VS-SAFE$(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$,
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

Transitions:

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

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

Figure 1: VS (modified version)

of [19]. We use the term *refinement* to denote a single-valued simulation relation.

3 The VS specification

In this paper we use a modified version of the group communication service, VS, defined in [12], and we refer the reader to the informal service description in that paper. The original VS service assumes that every processor in the universe $\mathcal{P}$ is a member of the initial view. In our setting the initial view is defined to be the distinguished initial view v_0, and we modify the specification of VS to reflect this fact. The modified specification is given in Figure 1. The fact that VS allows views to be created only in order of view identifier is unimportant: weakening this requirement to allow out-of-order view creation would not change the external behavior, because VS-NEWVIEW actions are constrained to occur in order anyway.

Invariant 3.1 (VS)
If $v, v' \in created$ and $v.id = v'.id$, then $v = v'$.

4 The DVS specification

Our DVS specification differs from the VS specification in the following ways: (1) DVS-REGISTER actions allow a client of the service to notify the service that it is ready to begin operating in a new view. This information is recorded in new variables, $registered[g]$, $g \in \mathcal{G}$. (2) New variables, $attempted[g]$, $g \in \mathcal{G}$, are introduced to remember which views have been reported to each process. (These are used in the proofs.) Also, new derived variables are introduced to remember which views have been attempted or registered at some member or all members. (3) The action DVS-CREATEVIEW only creates primary components, whereas the VS-CREATEVIEW is unconstrained (except for increasing ids). The specification is given in Figure 2. In this specification, $\mathcal{M}_c \subseteq \mathcal{M}$ denotes the set of messages that clients may use for communication.

The most interesting part of the DVS specification is the transition definition for DVS-CREATEVIEW(v). The precondition specifies the properties that a view must satisfy in order to be considered primary. For example, the precondition says that $v.set$ must intersect the membership set of all previously-created smaller-id views w for which there is no intervening totally registered view – that is, the set of all "possible previous primary views". Since (for convenience) we allow out of order view creation in DVS, we also include a symmetric condition for previously-created larger-id views.

DVS informs its clients of view changes using DVS-NEWVIEW actions. Even though views can be created out of view id order, the notification to each client is consistent with that order. Not every client needs to see every view. DVS allows the client at each processor p to "register" the current view at p with an action DVS-REGISTER$_p$. With this action, the client at p informs the service that it has obtained whatever information the application needs to begin operating in the new view. For many applications, this will mean that p has received messages from every other member, reporting its state at the start of the new view.

DVS allows a processor p to broadcast a message m using a DVS-GPSND$(m)_p$ action, and delivers the message to a processor q using a DVS-GPRCV$(m)_{p,q}$ action. DVS also uses a DVS-SAFE$(m)_{p,q}$ action to report to processor q that the earlier message m from p has been delivered to all members of the current view of q. DVS guarantees that messages sent by a processor p when the current view of p is v are delivered only within view v (i.e., only to processors in $v.set$ whose current view is v). Moreover, each processor receives messages in the same order as any other processor and without gaps in the sequence of received messages; however, some processors may receive only a prefix of the sequence of messages received by other processors.

Invariant 4.1 expresses the key intersection property guaranteed by DVS; this is weaker than the intersection property required by static definitions of primary views, which says that all primary components must intersect. This invariant

Signature:

Input: $\textsc{dvs-gpsnd}(m)_p$, $m \in \mathcal{M}_c$, $p \in \mathcal{P}$
$\textsc{dvs-register}_p$, $p \in \mathcal{P}$

Internal: $\textsc{dvs-createview}(v)$, $v \in \mathcal{V}$
$\textsc{dvs-order}(m, p, g)$, $m \in \mathcal{M}_c$, $p \in \mathcal{P}$, $g \in \mathcal{G}$

Output: $\textsc{dvs-gprcv}(m)_{p,q}$, $m \in \mathcal{M}_c$, $p, q \in \mathcal{P}$
$\textsc{dvs-safe}(m)_{p,q}$, $m \in \mathcal{M}_c$, $p, q \in \mathcal{P}$
$\textsc{dvs-newview}(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

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 $g \in \mathcal{G}$:
  $queue[g] \in seqof(\mathcal{M}_c \times \mathcal{P})$, init λ
  $attempted[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else
  $registered[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ 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
Derived variables:
  $\mathcal{A}tt \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid attempted[v.id] \neq \{\}\}$
  $\mathcal{T}ot\mathcal{A}tt \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid v.set \subseteq attempted[v.id]\}$
  $\mathcal{R}eg \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid registered[v.id] \neq \{\}\}$
  $\mathcal{T}ot\mathcal{R}eg \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid v.set \subseteq registered[v.id]\}$

Transitions:

internal $\textsc{dvs-createview}(v)$
Pre: $\forall w \in created : v.id \neq w.id$
  $\forall w \in created$:
    $\exists x \in \mathcal{T}ot\mathcal{R}eg : w.id < x.id < v.id$
    or $\exists x \in \mathcal{T}ot\mathcal{R}eg : v.id < x.id < w.id$
    or $v.set \cap w.set \neq \{\}$
Eff: $created := created \cup \{v\}$

output $\textsc{dvs-newview}(v)_p$
Pre: $v \in created$
  $v.id > current\text{-}viewid[p]$
Eff: $current\text{-}viewid[p] := v.id$
  $attempted[g] := attempted[g] \cup \{p\}$

input $\textsc{dvs-register}_p$
Eff: if $current\text{-}viewid[p] \neq \perp$ then
  $registered[current\text{-}viewid[p]] :=$
    $registered[current\text{-}viewid[p]] \cup \{p\}$

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

internal $\textsc{dvs-order}(m, p, g)$
Pre: m is head of $pending[p, g]$
Eff: remove head of $pending[p, g]$
  append $\langle m, p\rangle$ to $queue[g]$

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

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

Figure 2: DVS

is our version of the correctness requirement for dynamic view services that two consecutive primary views intersect.

Invariant 4.1 (DVS)
If $v, w \in created$, $v.id < w.id$, and there is no $x \in \mathcal{T}ot\mathcal{R}eg$ such that $v.id < x.id < w.id$, then $v.set \cap w.set \neq \{\}$.

Invariant 4.2 says that if a view w is totally attempted, then no earlier view v can still be "active".

Invariant 4.2 (DVS)
If $v \in created$, $w \in \mathcal{T}ot\mathcal{A}tt$, and $v.id < w.id$, then there exists $p \in v.set$ with $current\text{-}viewid[p] > v.id$.

5 An implementation of DVS

We now give an algorithm that implements the DVS service, in the sense of inclusion of sets of traces. We build the algorithm on top of the VS service and we use ideas from [18]. The overall system consists of an automaton VS-TO-DVS$_p$ for each $p \in \mathcal{P}$, and VS.

5.1 The implementation

The automaton VS-TO-DVS$_p$ is given in Figure 3. VS-TO-DVS$_p$ uses special non-client messages, tagged either with *"info"* or *"registered"*. Thus, we use $\mathcal{M} = \mathcal{M}_c \cup (\{\text{"info"}\} \times \mathcal{V} \times 2^{\mathcal{V}}) \cup \{\text{"registered"}\}$, where $\mathcal{M}_c$ is the set of all client messages and $\mathcal{M}$ is the universe of all messages. The *attempted*, *reg*, and *info-sent* state variables are not needed for the algorithm, but only for the proofs.

VS-TO-DVS$_p$ acts as a "filter", receiving VS-NEWVIEW inputs from the underlying VS service and deciding whether to accept the proposed views as primary views. If VS-TO-DVS$_p$ decides to accept some such view v, it "attempts" the view by performing a DVS-NEWVIEW(v) output. For each v, we think of the DVS internal DVS-CREATEVIEW(v) action as occurring at the time of the first DVS-NEWVIEW(v) event.

According to the DVS specification, the algorithm is supposed to guarantee nonempty intersection of each newly-created primary view v with any previously-created view w having no intervening totally registered view – a *global* condition involving *nonempty* intersection. The VS-TO-DVS$_p$ processors, however, do not have accurate knowledge of which primary views have been created by other processors, nor of which views are totally registered. Therefore, the processors employ a *local* check of *majority* intersection with known views, rather than a global check of nonempty intersection with existing views. Specifically, each VS-TO-DVS$_p$ keeps track of an "active" view *act*, which is the latest view that it knows to be totally registered, plus a set of "ambiguous" views *amb*, which are all the views that it knows have been attempted (i.e., have had a DVS-NEWVIEW action performed someplace), and whose ids are greater than *act.id*. We define $use = \{act\} \cup amb$. When VS-TO-DVS$_p$ receives a VS-NEWVIEW(v) input, it sends out *"info"* messages containing its current *act* and *amb* values to all the other processors in the new view, using the VS service, and then waits to receive corresponding *"info"* messages for view v from all the other processors in the view. After receiving this information

Signature:

Input: DVS-GPSND$(m)_p$, $m \in \mathcal{M}_c$
DVS-REGISTER$_p$
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
  $msgs\text{-}from\text{-}vs[g] \in seqof(\mathcal{M}_c \times P)$, init λ
  $safe\text{-}from\text{-}vs[g] \in seqof(\mathcal{M}_c \times P)$, init λ
  $reg[g]$ a bool, init `true` if $p \in P_0$ and $g = g_0$, `false` else
  $info\text{-}sent[g] \in (\mathcal{V} \times 2^{\mathcal{V}})_\perp$, init $\perp$

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 $use = \{act\} \cup amb$

Transitions:

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: $info\text{-}rcvd[q, cur.id] := \langle v, V\rangle$
  if $v.id > act.id$ then $act := v$
  $amb := \{w \in amb \cup V \mid w.id > act.id\}$

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$_p$
Eff: if $client\text{-}cur \neq \perp$ then
  $reg[client\text{-}cur] :=$ `true`
  append $\langle$ *"registered"*$\rangle$ to $msgs\text{-}to\text{-}vs[client\text{-}cur.id]$

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: $act := v$
  $amb := \{w \in amb \mid w.id > act.id\}$

input DVS-GPSND$(m)_p$
Eff: if $client\text{-}cur.id_p \neq \perp$ then
  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 3: VS-TO-DVS$_p$

(and updating its own *act* and *amb* accordingly), VS-TO-DVS$_p$ checks that v has a majority intersection with each view in *use*. If so, VS-TO-DVS$_p$ performs a DVS-NEWVIEW$_p$ output.

Then the clients can use the communication system to exchange state information as needed for processing in view v. When client at p has obtained enough information, it "registers" the view by means of action DVS-REGISTER$_p$, which causes processor p to send *"registered"* messages to the other members. When a processor receives *"registered"* messages for a view v from all members, it may perform garbage collection by discarding information about views with ids smaller than that of v. VS-TO-DVS uses VS to send and receive messages.

We define the system DVS-IMPL to be the composition of all the VS-TO-DVS$_p$ automata and VS with all the external actions of VS hidden. We introduce four derived variables for DVS-IMPL analogous to those of DVS, indicating the attempted, totally attempted, registered, and totally registered views, respectively. They are:
$\mathcal{A}tt = \{v \in created \mid (\exists p \in v.set) v \in attempted_p\}$;
$\mathcal{T}ot\mathcal{A}tt = \{v \in created \mid (\forall p \in v.set) v \in attempted_p\}$;
$\mathcal{R}eg = \{v \in created \mid (\exists p \in v.set) reg[v.id]_p = \texttt{true}\}$; and
$\mathcal{T}ot\mathcal{R}eg = \{v \in created \mid (\forall p \in v.set) reg[v.id]_p = \texttt{true}\}$.

5.2 Invariants

This section contains the main invariants of DVS-IMPL needed for the refinement proof in Section 5.3.

Invariant 5.1 (DVS-IMPL)
If $v \in attempted_p$ and $q \in v.set$ then $cur.id_q \geq v.id$.

Invariant 5.2 (DVS-IMPL)
1. $act_p \in \mathcal{T}ot\mathcal{R}eg$.
2. *If $w \in amb_p$ then $act.id_p < w.id$.*
3. *If $client\text{-}cur_p \neq \perp$ and $w \in \{act_p\} \cup amb_p$, then $w.id \leq client\text{-}cur.id_p$.*
4. *If $info\text{-}sent[g]_p = \langle x, X\rangle$ then $x \in \mathcal{T}ot\mathcal{R}eg$.*
5. *If $info\text{-}sent[g]_p = \langle x, X\rangle$ and $w \in X$ then $x.id < w.id$.*
6. *If $info\text{-}sent[g]_p = \langle x, X\rangle$ and $w \in \{x\} \cup X$ then $w.id < g$.*

Invariant 5.3 says that certain views appear in *"info"* messages, unless they have been garbage-collected.

Invariant 5.3 (DVS-IMPL)
1. If $info\text{-}sent[g]_p = \langle x, X\rangle$ and $w \in attempted_p$, then either $w \in \{x\} \cup X$ or $w.id < x.id$.
2. If $info\text{-}rcvd[q, g]_p = \langle x, X\rangle$ and $w \in \{x\} \cup X$, then either $w \in use_p$ or $w.id < act.id_p$.

Invariant 5.4 says that two attempted views having no intervening totally registered view, and having a common member, q, that has attempted the first view, must intersect in a majority of processors. This is because, under these circumstances, information must flow from q to any process that attempts the second view.

Invariant 5.4 (DVS-IMPL)
Suppose that $v \in attempted_p$, $q \in v.set$, $w \in attempted_q$, $w.id < v.id$, and there is no $x \in \mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. Then $|v.set \cap w.set| > |w.set|/2$.

Proof: By induction on the length of an execution.
Base: In the initial state, only v_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.
Inductive step: Fix any step (s, π, s'), where s is reachable, and assume the invariant is true in state s. We show that it is true in s'. So fix v, w, p, and q, and assume that $v \in s'.attempted_p$, $q \in v.set$, $w \in s'.attempted_q$, $w.id < v.id$, and there is no $x \in s'.\mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. Then also there is no $x \in s.\mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. We consider four cases:
1. $v \in s.attempted_p$ and $w \in s.attempted_q$. Then the statement for s implies that $|v.set \cap w.set| > |w.set|/2$, as needed.
2. $v \notin s.attempted_p$ and $w \notin s.attempted_q$. This cannot happen because we cannot have both v and w becoming attempted in a single step.
3. $v \notin s.attempted_p$ and $w \in s.attempted_q$. Then π must be DVS-NEWVIEW$(v)_p$. By the precondition of π we have that $s.info\text{-}rcvd[q, v.id]_p = \langle x, X\rangle$ for some x and X. It follows that $s.info\text{-}sent[v.id]_q = \langle x, X\rangle$. Then Invariant 5.3 implies that either $w \in \{x\} \cup X$ or $w.id < x.id$. If $w.id < x.id$, then we obtain a contradiction, because $x \in s.\mathcal{T}ot\mathcal{R}eg$ (by Invariant 5.2) and $x.id < v.id$ (by Invariant 5.2). So $w \in \{x\} \cup X$. Now by Invariant 5.3 we have that either $w \in s.use_p$ or $w.id < s.act.id_p$. In the former case, by the precondition of π, we have $|v.set \cap w.set| > |w.set|/2$. In the latter case, we obtain a contradiction, because $s.act_p \in \mathcal{T}ot\mathcal{R}eg$ (by Invariant 5.2) and Invariant 5.2 implies that $s.act.id_p \leq s.client\text{-}cur_p < s.cur_p = v.id$. 4. $v \in s.attempted_p$ and $w \notin s.attempted_q$. Then π must be DVS-NEWVIEW$(w)_q$. But this cannot happen: Since $v \in s.attempted_p$ and $q \in v.set$, Invariant 5.1 implies that $s.cur.id_q \geq v.id$. Since $v.id > w.id$, we have $s.cur.id_q > w.id$. But the precondition of action π requires $s.cur.id_q = w.id$, so π is not enabled in s. ∎

Invariant 5.5 says that any attempted view v intersects the latest preceding totally registered view w in a majority of members of w.

Invariant 5.5 (DVS-IMPL)
Suppose that $v \in \mathcal{A}tt$, and $w \in \mathcal{T}ot\mathcal{R}eg$, $w.id < v.id$, and there is no $x \in \mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. Then $|v.set \cap w.set| > |w.set|/2$.

Proof: By induction on the length of an execution.
Base: In the initial state, fix v and w satsifying the hypotheses. The first two assumptions imply that $v = w = v_0$. But then the third assumption $w.id < v.id$ is false. Thus, the hypothesis of the statement is false, so the statement is vacuously true.
Inductive step: Fix any step (s, π, s'), where s is reachable, assume the invariant is true in state s, and show that it is true in s'. So fix v and w, and assume that $v \in s'.\mathcal{A}tt$, $w \in s'.\mathcal{T}ot\mathcal{R}eg$, $w.id < v.id$, and there is no $x \in s'.\mathcal{T}ot\mathcal{R}eg, w.id < x.id < v.id$. We consider four cases:
1. $v \in s.\mathcal{A}tt$ and $w \in s.\mathcal{T}ot\mathcal{R}eg$. Then the statement for s implies that $|v.set \cap w.set| > |w.set|/2$, as needed.
2. $v \notin s.\mathcal{A}tt$ and $w \notin s.\mathcal{T}ot\mathcal{R}eg$. This cannot happen because we cannot have both v becoming attempted and w becoming totally registered in a single step.
3. $v \notin s.\mathcal{A}tt$ and $w \in s.\mathcal{T}ot\mathcal{R}eg$. Then π must be DVS-NEWVIEW$(v)_p$ for some p. The precondition of π implies that, for any view $y \in s.use_p$, $|v.set \cap y.set| > |y.set|/2$. Hence to prove the claim it is enough to prove that $w \in s.use_p$. We proceed by contradiction assuming that $w \notin s.use_p$.
By Invariant 5.2, $s.use_p \cap s.\mathcal{T}ot\mathcal{R}eg \neq \{\}$. Let m be the view in $s.\mathcal{T}ot\mathcal{R}eg \cap s.use_p$ having the biggest identifier. We know that $m \neq w$ because $w \notin s.use_p$. It follows then that $m.id \neq w.id$. We claim that $m.id < w.id$. Suppose for the sake of contradiction that $m.id > w.id$. Since $m \in s.use_p$, Invariant 5.2 implies that $m.id \leq s.client\text{-}cur_p < s.cur_p = v.id$. So $w.id < m.id < v.id$. Since $m \in s'.\mathcal{T}ot\mathcal{R}eg$, this contradicts the hypothesis of the inductive step. Therefore, $m.id < w.id$. Let n be the view in $s.\mathcal{T}ot\mathcal{R}eg$ that has the smallest id strictly greater than that of m. Note that $m.id < n.id \leq w.id < v.id$. Since $m \in s.use_p$, the precondition of π implies that $|v.set \cap m.set| > |m.set|/2$. By the statement applied to state s, $|n.set \cap m.set| > |m.set|/2$. Hence there exists a processor $q \in v.set \cap n.set$. By the precondition of π, $s.info\text{-}rcvd[q, v.id]_p = \langle x, X\rangle$ for some x, X. Invariant 5.2 implies that $x.id < v.id$. Since $n \in s.\mathcal{T}ot\mathcal{R}eg$, we have that $n \in s.attempted_q$. Then Invariant 5.3 (used with $w = n$) implies that either $w \in \{x\} \cup X$ or $w.id < x.id$. In either case, $\{x\} \cup X$ contains a view $y \in s.\mathcal{T}ot\mathcal{R}eg$ (either w or x) such that $n.id \leq y.id < v.id$. Then Invariant 5.3 implies that either $y \in s.use_p$ or $y.id < s.act.id_p$. By Invariant 5.2, $s.act_p \in s.\mathcal{T}ot\mathcal{R}eg$ and by definition, $s.act_p \in s.use_p$. So in either case, the hypothesis that m is the totally registered view with the largest id belonging to $s.use_p$ is contradicted.
4. $v \in s.\mathcal{A}tt$ and $w \notin s.\mathcal{T}ot\mathcal{R}eg$. Then π must be DVS-REGISTER$_p$ for some p. Let m be the view in $s.\mathcal{T}ot\mathcal{R}eg$ with the largest id that is strictly less than $w.id$. By the statement for s, we know that $|w.set \cap m.set| > |m.set|/2$ and $|v.set \cap m.set| > |m.set|/2$. Hence there is a processor $q \in w.set \cap v.set$.
Since $v \in s.\mathcal{A}tt$, there exists a processor r such that $v \in s.attempted_r$; thus also $v \in s'.attempted_r$. Since $w \in s'.\mathcal{T}ot\mathcal{R}eg$, we have that $w \in s'.attempted_q$. By assumption, there is no view $x \in s'.\mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. By Invariant 5.4 applied to state s' (with $p = r$), we have that $|v.set \cap w.set| > |w.set|/2$, as needed. ∎

The final invariant, a corollary to Invariant 5.5, is instrumental in the refinement proof.

Invariant 5.6 (DVS-IMPL)
If $v, w \in \mathcal{A}tt$, $w.id < v.id$, and there is no $x \in \mathcal{T}ot\mathcal{R}eg$ with $w.id < x.id < v.id$, then $v.set \cap w.set \neq \{\}$.

Proof: Suppose that v and w are as given. We consider two cases.
1. $w \in \mathcal{T}ot\mathcal{R}eg$. Then since there is no $x \in \mathcal{T}ot\mathcal{R}eg$ with $w.id < x.id < v.id$, it follows that w is the view with the largest id in the set $\{y \in \mathcal{T}ot\mathcal{R}eg : y.id < v.id\}$. Then Invariant 5.5 implies that $|v.set \cap w.set| > |w.set|/2$, which implies that $v.set \cap w.set \neq \{\}$, as needed.
2. $w \notin \mathcal{T}ot\mathcal{R}eg$. Then let $Y = \{y \in \mathcal{T}ot\mathcal{R}eg : y.id < w.id\}$. We claim that Y is nonempty: We know that $v_0 \in \mathcal{T}ot\mathcal{R}eg$ and that $v_0.id \leq w.id$. If $v_0.id = w.id$, then by Lemma 3.1, we

have $w = v_0$. But then $w \in \mathcal{TotReg}$, a contradiction. So we must have $v_0.id < w.id$, which implies that $v_0 \in Y$, so Y is nonempty.
Now fix z to be the view in Y with the largest id. Since there is no $x \in \mathcal{TotReg}$ with $w.id < x.id < v.id$, it follows that z is also the view with the largest id in the set $\{y \in \mathcal{TotReg} : y.id < v.id\}$. Then Invariant 5.5 implies that $|w.set \cap z.set| > |z.set|/2$ and $|v.set \cap z.set| > |z.set|/2$. Together, these two facts imply that $v.set \cap w.set \neq \{\}$, as needed. ∎

5.3 The refinement

We prove that DVS-IMPL implements DVS by defining a function $\mathcal{F}$ that maps states of DVS-IMPL to states of DVS and proving that this function is a *refinement*.

DVS-IMPL uses VS to send client messages and messages generated by the implementation (*"info"* and *"registered"* messages). The refinement discards the non-client messages. Thus, if q is a finite sequence of client and non-client messages, we define $purge(q)$ to be the queue obtained by deleting any *"info"* or *"registered"* messages from q, and $purgesize(q)$ to be the number of *"info"* and *"registered"* messages in q.
Figure 4 defines the refinement $\mathcal{F}$. The fact that $\mathcal{F}$ is a refinement is shown using two lemmas, 5.7 and 5.8, expressing the two conditions required by the definition of a refinement:
Lemma 5.7 *If s is an initial state of* DVS-IMPL *then $\mathcal{F}(s)$ is an initial state of* DVS.

Lemma 5.8 *Let s be a reachable state of* DVS-IMPL*, $\mathcal{F}(s)$ a reachable state of* DVS*, and (s, π, s') a step of* DVS-IMPL*. Then there is an execution fragment α of* DVS *that goes from $\mathcal{F}(s)$ to $\mathcal{F}(s')$, such that $trace(\alpha) = trace(\pi)$.*

Proof: By case analysis based on the type of the action π. The only interesting case is where $\pi =$ DVS-NEWVIEW$(v)_p$. Then $trace((s, \pi, s)) = \pi$. Define $t = \mathcal{F}(s)$ and $t' = \mathcal{F}(s')$. We consider two cases:
1. $v \in t.created$. In this case, we set $\alpha = (t, \pi', t')$, where $\pi' =$ DVS-NEWVIEW$(v)_p$. The code shows that π' brings DVS from state t to state t'. It remains to prove that π' is enabled in state t, that is, that $v \in t.created$ and $v.id > t.current\text{-}viewid[p]$. The first of these two conditions is true because of the defining condition for this case. The second condition follows from the precondition of π in DVS-IMPL: this precondition implies that $v.id > s.client\text{-}cur.id_p$, and by the definition of $\mathcal{F}$ we have $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$.
2. $v \notin t.created$. In this case we set $\alpha = (t, \pi', t'', \pi'', t')$, where $\pi' =$ DVS-CREATEVIEW$(v)_p$, $\pi'' =$ DVS-NEWVIEW$(v)_p$, and t'' is the unique state that arises by running the effect of π' from t. The code shows that α brings DVS from state t to state t'. It remains to prove that π' is enabled in t and that π'' is enabled in t''.
The precondition of π' requires that (i) $\forall w \in t.created$, $v.id \neq w.id$ and (ii) $\forall w \in t.created$, either $\exists x \in s.\mathcal{TotReg}$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$, or else $v.set \cap w.set \neq \{\}$. To see requirement (i), suppose for the sake of contradiction that $w \in t.created$ and $w.id = v.id$. The precondition of π in DVS-IMPL implies that $v = s.cur_p$, which implies that $v \in s.created$. Since $w \in t.created$, the definition of $\mathcal{F}$ implies that $w \in s.attempted_q$ for some q. This implies that $w \in s.created$. But then Invariant 3.1 implies that $v = w$. But this contradicts that fact that $v \notin t.created$ and $w \in t.created$. To see requirement (ii), suppose that $w \in t.created$ and there is no $x \in s.\mathcal{TotReg}$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$. Then $w \in s.attempted_q$ for some q, by definition of $\mathcal{F}$. Part (i) implies that $w \neq v$, so also $w \in s'.attempted_q$. Therefore, $w \in s'.\mathcal{A}tt$. We also have $v \in s'.\mathcal{A}tt$. Moreover, there is no $x \in s'.\mathcal{TotReg}$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$. Then Invariant 5.6 implies that $v.set \cap w.set \neq \{\}$, as needed to prove that π' is enabled in t.
We now prove that π'' is enabled in state t''. The precondition of π'' requires that $v \in t''.created$ and $v.id > t''.current\text{-}viewid[p]$. The first condition is true because v is added to *created* by π'. The second condition follows from the precondition of π in DVS-IMPL: The precondition of π implies that $v.id > s.client\text{-}cur.id_p$. The definition of $\mathcal{F}$ implies that $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$. Moreover, $t''.current\text{-}viewid[p] = t.current\text{-}viewid[p]$. It follows that $v.id > t''.current\text{-}viewid[p]$. Thus π'' is enabled in state t''.
∎

Theorem 5.9 *Every trace of* DVS-IMPL *is a trace of* DVS.

Proof: Lemmas 5.7 and 5.8 imply that $\mathcal{F}$ is a refinement, which implies the result. ∎

6 An application of DVS

Now we show how to use DVS to implement the totally ordered broadcast service TO, defined in [12]. This service accepts messages from clients and delivers them to all clients according to the same total order. The implementation consists of an automaton DVS-TO-TO$_p$ for each $p \in \mathcal{P}$, and the DVS specification.

6.1 The implementation

The implementation is very similar to the TO implementation provided in [12]; we refer the reader to the informal algorithm description in that paper. Both algorithms rely on primary views to establish a total order of client messages. The main difference is that the algorithm in [12] uses a static notion of primary and the new one uses a dynamic notion. The algorithm of [12] is built upon a VS service that reports non-primary as well as primary views, and uses a simple local test to determine if the view is primary. That algorithm does some non-critical background work (gossiping information) in non-primary views. In contrast, the new algorithm is built upon our DVS service, which only reports primary views. Thus the new algorithm is simpler in that it does not perform the local tests and does not carry out any processing in non-primary views. On the other hand, in the new algorithm, the application programs must perform DVS-REGISTER actions to tell the DVS service when they have "established" new views. Although the new algorithm appears very similar to the old one, the fact that the DVS service provides weaker and more complicated guarantees than the VS service makes the new algorithm harder to prove correct. Another, minor difference between this algorithm and that of [12] is the new handling of initial views. In particular, a *delay* buffer is used for client messages, to accommodate messages that might arrive before a node has any view.

The code for automaton DVS-TO-TO$_p$ appears in Figure 5. In this code, $\mathcal{L} = \mathcal{G} \times \mathbf{N}^{>0} \times \mathcal{P}$ is the set of *labels*, with selectors $l.id$, $l.seqno$ and $l.origin$. $\mathcal{A}$ is the set of messages that can be sent by the clients of the TO service. $\mathcal{C} = \mathcal{L} \times \mathcal{A}$ is the set of possible associations between labels and client messages. $\mathcal{S} = 2^{\mathcal{C}} \times seqof(\mathcal{L}) \times \mathbf{N}^{>0} \times \mathcal{G}$ is the set of *summaries*, with selectors $x.con$, $x.ord$, $x.next$ and $x.high$.

If Y is a partial function from processor ids to summaries, then we define:
$knowncontent(Y) = \cup_{q \in dom(Y)} Y(q).con$,
$maxprimary(Y) = \max_{q \in dom(Y)}\{Y(q).high\}$,
$maxnextconfirm(Y) = \max_{q \in dom(Y)} Y(q).next$,

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

- $t.created = \cup_{p \in \mathcal{P}} s.attempted_p$
- 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 $p \in \mathcal{P}$, $g \in \mathcal{G}$, $t.pending[p, g] = purge(s.pending[p, g]) + purge(s.msgs\text{-}to\text{-}vs[g]_p)$
- for each $g \in \mathcal{G}$, $t.queue[g] = purge(s.queue[g])$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$,
  $t.next[p, g] = s.next[p, g] - purgesize(s.queue[g](1..next[p, g] - 1)) - |s.msgs\text{-}from\text{-}vs[g]_p|$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$,
  $t.next\text{-}safe[p, g] = s.next\text{-}safe[p, g] - purgesize(s.queue[g](1..next\text{-}safe[p, g] - 1)) - |s.safe\text{-}from\text{-}vs[g]_p|$

Figure 4: The refinement $\mathcal{F}$.

$reps(Y) = \{q \in dom(Y) : Y(q).high = maxprimary\}$,
$chosenrep(Y)$ = some element in $reps(Y)$,
$shortorder(Y) = Y(chosenrep(Y)).ord$, and
$fullorder(Y) = shortorder(Y)$ followed by the remaining elements of $dom(knowncontent(Y))$, in label order.

The algorithm involves *normal* and *recovery* activity. Normal activity occurs while a group view is not changing. Recovery activity begins when a new primary view is presented by DVS, and continues while the members combine information from their previous history, to provide a consistent basis for ongoing normal activity.

During normal activity, each client message received by TO-IMPL is given a system-wide unique *label*, which is remembered in a relation *content* and communicated to other processes in the same view using DVS. When a message is received, the label is given an *order*, a tentative position in the system-wide total order the service is to provide. The consistent sequence of message delivery within each view keeps this tentative order consistent at members of a given view, but it need not always be consistent between nodes in different views. When all members of a view have given a label an order, the label and its order may become *confirmed.* This is deduced by the node from the occurrence of the safe indication for the message that carried the label. The messages associated with confirmed labels may be released to the clients in the given order.

When a new primary view is reported by DVS, recovery activity occurs to integrate the knowledge of different members. First, each member of a new view sends a message, using DVS, that contains a summary of that node's state, including the tentative order built in its previous view. Once a node has received all members' state summaries, it processes the information in one atomic step, i.e., it *establishes* the new view. Once a node establishes a view, it informs DVS of that fact with a DVS-REGISTER action. Then recovery continues by collecting the DVS safe indications. Once the state exchange is safe, all labels used in the exchange are marked as safe and all associated messages are confirmed just as in normal processing.

The system TO-IMPL is the composition of all the DVS-TO-TO$_p$ automata and DVS with all the external actions of DVS hidden. The *allstate* and *allconfirm* derived variables are defined for TO-IMPL just as in [12].

6.2 Correctness proof

The correctness proof for TO-IMPL follows the outline of the one in [13]. The main difference is that the main invariant, which corresponds to Lemma 6.17 of [12], requires a different, more subtle proof. Invariant 6.1 says that any view that is known (anywhere in the system state) to be an established primary was in fact attempted by all its members.

Invariant 6.1 (TO-IMPL)
If $x \in allstate$ then there exists $w \in created$ such that $x.high = w.id$, and for all $p \in w.set$, $p \in attempted[w.id]$.

Invariant 6.2 says that if a view w is established, then no earlier view v can still be active.

Invariant 6.2 (TO-IMPL)
If $v \in created$, $x \in allstate$ and $x.high > v.id$ then there exists $p \in v.set$ with $current.id_p > v.id$.

Proof: Fix v, x as given. Lemma 6.1 shows the existence of $w \in created$ such that $x.high = w.id$, and for all $p \in w.set$, $p \in attempted[w.id]$. Then Invariant 4.2 implies that there exists $p \in v.set$ with $current\text{-}viewid[p] > v.id$. But $current\text{-}viewid[p] = current.id_p$, which yields the result. ∎

Finally we provide the proof for the invariant corresponding to the invariant stated in Lemma 6.17 of [13].

Invariant 6.3 (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 $x.high > v.id$, $\sigma \leq x.ord$.

Proof: By induction on the length of the execution. We present the differences from the proof in [13].
Base: In the initial state, the only created view is v_0, and there is no $x \in allstate$ with $x.high > g_0$. So the statement is vacuously true.
Inductive step: Fix any step (s, π, s'), where s is reachable, assume the invariant is true in state s, and show that it is true in s'. So fix $v \in s'.created$ and σ, and assume that for every $p \in v.set$, if $s'.current.id_p > v.id$ then $s'.established[v.id]_p$ and $\sigma \leq s'.buildorder[p, v.id]$.

If $v \notin s.created$, then π must be CREATEVIEW(v). Then $s'.established[v.id]_p = false$ for all p. Fix $x \in s'.allstate$ and suppose that $x.high > v.id$. Then Invariant 6.2 applied to s' implies that there exists $p \in v.set$ with $s'.current.id_p > v.id$; fix such a p. Then the hypothesis part of the invariant for s' implies that $s'.established[v.id]_p = true$, a contradiction. It follows that $v \in s.created$.

As before, the interesting steps are GPRCV$_p$ steps that produce a new summary x by delivering the last state-exchange message of a view w to some processor p. Thus $x.high = w.id$. Let x' be the summary of $q' = chosenrep$ in $s'.gotstate$. We claim $x'.high \geq v.id$.

To prove the claim, we let v' denote the unique element with highest viewid among the elements of $s'.created$ such

Signature:

Input: $\textsc{bcast}(a)_p$, $a \in \mathcal{A}$
$\textsc{dvs-gprcv}(m)_{q,p}$, $q \in \mathcal{P}$, $m \in \mathcal{C} \cup \mathcal{S}$
$\textsc{dvs-safe}(m)_{q,p}$, $q \in \mathcal{P}$, $m \in \mathcal{C} \cup \mathcal{S}$
$\textsc{dvs-newview}(v)_p$, $v \in \mathcal{V}$

Output: $\textsc{dvs-register}_p$
$\textsc{dvs-gpsnd}(m)_p$, $m \in \mathcal{C} \cup \mathcal{S}$
$\textsc{brcv}(a)_{q,p}$, $a \in \mathcal{A}$, $q \in \mathcal{P}$

Internal: $\textsc{confirm}_p$

State:

$current \in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else
$status \in \{normal, send, collect\}$, init $normal$
$content \in \mathcal{C}$, 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
$highprimary \in \mathcal{G}$, init g_0
$gotstate$, a partial function from $\mathcal{P}$ to $\mathcal{S}$, init $\{\}$
$safe\text{-}exch \subseteq \mathcal{P}$, init $\{\}$
$registered \subseteq \mathcal{G}$, init $\{g_0\}$ if $p \in P_0$, $\{\}$ else
$delay \in seqof(\mathcal{A})$, init λ
for each $g \in \mathcal{G}$,
  $established[g]$, a bool, init `false`

Transitions:

input $\textsc{bcast}(a)_p$
 Eff: append a to $delay$

internal $\textsc{label}(a)_p$
 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 $\textsc{dvs-gpsnd}(\langle l, a\rangle)_p$
 Pre: $status = normal$
  l is head of $buffer$
  $\langle l, a\rangle \in content$
 Eff: delete head of $buffer$

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

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

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

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

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

output $\textsc{dvs-gpsnd}(x)_p$
 Pre: $status = send$
  $x = \langle content, order, nextconfirm, highprimary\rangle$
 Eff: $status := collect$

input $\textsc{dvs-gprcv}(x)_{q,p}$
 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 $\textsc{dvs-register}_p$
 Pre: $current \neq \perp$
  $established[current]$
  $current \notin registered$
 Eff: $registered := registered \cup \{current\}$

input $\textsc{dvs-safe}(x)_{q,p}$
 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 5: $\textsc{dvs-to-to}_p$

that $v'.id < w.id$ and $s'.registered[v'.id] = v'.set$. Let v'' denote either v' or v, whichever has the higher viewid. Invariant 4.1 shows that $w.set \cap v''.set \neq \{\}$, no matter whether v'' is v or v'. Fix any element q'' in $w.set \cap v''.set$. Recall that the condition for establishing a view shows that $domain(s'.gotstate_p) = w.set$, so by the code, either $q'' \in domain(s.gotstate_p)$ or else q'' is the sender of the message whose receipt is the step we are examining. In the former case, let x'' be the summary $s.gotstate(q'')_p$; in the latter let x'' be the summary whose receipt is the event. In either case we have $x'' \in s.allstate[q'', w.id]$.

We now show that $s.established[v''.id]_{q''}$. We consider two cases:
1. $v'' = v'$. Then $q'' \in v'.set$ so by definition of v', we obtain $q'' \in s.registered[v'.id]$; therefore, $s.established[v'.id]_{q''}$.
2. $v'' = v$. Because $s.allstate[q'', w.id]$ is non-empty, the analogue of part 4 of Lemma 6.6 from [13] implies that $s.current.id_{q''} \geq w.id$. We have that $x.high > v.id$ by assumption, and $x.high = w.id$ by the code; therefore, $w.id > v.id$. So also $s.current.id_{q''} > v.id$. Recall that we are in the case where the hypothesis of this lemma is true. Therefore, by this hypothesis (uses $q'' \in v.set$), we obtain that $s.established[v.id]_{q''}$

By the analogue of Lemma 6.13 from [13], (applied with q'' replacing p) we obtain $x''.high \geq v''.id$. By the definition of q' as a member that maximizes the *high* component in the summary recorded in $s'.gotstate$, we have $x'.high \geq x''.high$. Therefore $x'.high \geq v''.id \geq v.id$, completing our proof of the claim. The rest of the proof is as before. ∎

To complete the implementation proof, we define a function from the reachable states of TO-IMPL to the states of TO and prove that it is a refinement. This function is defined exactly as in [12], except that, for each p, the abstract pending queue $t.pending[p]$ is defined to include an additional tail consisting of the contents of $s.delay_p$.

Theorem 6.4 *Every trace of* TO-IMPL *is a trace of* TO.

7 Discussion

This work deals entirely with safety properties; it remains to consider performance and fault-tolerance properties as well. It also remains to study other applications of our DVS specification, such as replicated data applications and load-balancing applications.

Another interesting exploration direction considers variations on the DVS specification, for example, one in which the state exchange at the beginning of a new view is supported by the dynamic view service. We are currently studying variations on our specifications that are more specifically tuned to systems like Isis and Ensemble. In particular, we would like to understand the power of the Isis requirement that processes that move together from one view to the next receive exactly the same messages in the first view, especially for coherent-data applications.

It would also be interesting to generalize the DVS service to dynamic *sets of primaries* rather than individual primaries, in order to allow tolerance of transient failures during a particular view.

Acknowledgments: We thank Ken Birman, who urged us to consider the interesting issues of dynamic views. We also thank Idit Keidar and Robbert van Renesse for discussions about our DVS specification and our algorithm models and proofs.

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] A. Fekete, N. Lynch and A. Shvartsman "Specifying and using a partitionable group communication service", extended version of [12], available at `http://theory.lcs.mit.edu/tds`.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

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