A Dynamic Primary View Group Communication Service[*]

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

March 3, 2002

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. The specification is a simple automaton, with only seven kinds of actions. We demonstrate the value of DVS 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 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, in the sense of trace inclusion.

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

Applications designed for distributed systems must cope with failures, because in practical settings failures are very likely to happen in a distributed system. Coping with failures in a distributed system, however, is not an easy task. A convenient approach is that of using general purpose building

[*] This research was supported by the following contracts: ARPA F19628-95-C-0118, AFOSR F49620-97-1-0337, NSF 9225124-CCR, and NSF ITR 0121277.

[†] Dipartimento di Informatica ed Applicazioni, Università di Salerno, 84081 Baronissi (SA), Italy. This author is also a member of Akamai's Office of Strategy and Technology.

[‡] Basser Department of Computer Science, Madsen Building F09, University of Sydney, NSW 2006, Australia.

[§] Dept. of Computer Science and Eng., 191 Auditorium Rd., Unit 3155, University of Connecticut, Storrs, CT 06269, USA and MIT Laboratory for Computer Science, 545 Technology Square, NE43-371, Cambridge, MA 02139, USA. The work of this author was in part supported by a NSF CAREER Award and by the NSF Grant 9988304.

blocks that provide powerful distributed services and facilitate the construction of applications. One such building block is a *view-oriented group communication service.*

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. Examples of view-oriented group communication services are found in Isis [5], Transis [13], Totem [31], Newtop [16], Relacs [2], and Horus [34].

Solutions to practical, real world problems have benefited from group communication services. Isis-based software has been used to provide reliable group communication services for the New York Stock Exchange, for the Swiss Electronic Bourse and for the French Air Traffic Control System [6].

The heart of a group communication service is 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. Views are crucial because they describe which processes participate in the computation and the system allow them to cooperate by guaranteeing that 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. Clearly each particular group communication service has its own set of properties that are offered to the user. A good survey of group communication services that provides a description of the guarantees made by each service is provided in [37].

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 about safety, performance, and fault-tolerance. Examples of specifications for group membership services and view-oriented group communication services appear in [3, 4, 7, 9, 14, 17, 18, 19, 20, 30, 32, 35, 36].

In [17], we presented a specification, 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 and fault-tolerance requirements. We used this specification as the basis for proving the correctness of a complex totally-ordered-broadcast algorithm based on [22, 1]. In ensuing work, Chockler has used a version of VS to model and verify an adaptive totally-ordered-broadcast algorithm [8], Lesley and Fekete [25] have proved that a version of an algorithm of Cristian and Schmuck [10] implements VS, and Khazan [23, 24] has used VS in the design of a load-balancing database algorithm.

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 pre-defined 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 [12, 15, 21, 26, 33].

In particular, Lotem, Keidar, and Dolev [26] 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 [26] 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 primary view group communication service. This service is inspired by the implementation of Lotem et al., but integrates communication with the group membership service. 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, typically, an exchange of information, 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. We expect each application process to indicate when it has completed this pre-processing for a new view v by "registering" the view. 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. 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.

Another feature of our specification, compared to that in [26], 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.

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. 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 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 [22, 1, 17], 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 we think 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 [36] is expressed in terms of 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. [26] use a property of an execution, which was previously defined by Cristian [9] for majority groups: any two (primary) views are linked 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 properties of applications running above them.

Our specification omits some properties of existing dynamic primary view management algorithms. For example, Isis [5] guarantees that processes that move together from one view to the next receive exactly the same messages in the first view. Guaranteeing this property requires state exchange within the view management service. This property is not needed to verify properties of applications such as the one giving a totally-ordered broadcast. Also, our service provides no explicit support for application-level state exchange. Systems like Isis do provide such support, by allowing application-level state exchange messages to be piggybacked on the lower-level state exchange messages.

In Section 2 we present our mathematical notation. The DVS service is presented in Section 3, and its implementation in Section 4. In Section 5 we use DVS to implement a totally ordered broadcast service. Section 6 contains some conclusions.

2 Mathematical foundations

2.1 Sets, functions, sequences

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 $S \cup \{\perp\}$. Whenever S is ordered, we order $S_\perp$ by extending the order on S, and making $\perp$ less than all elements of S. 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*.

2.2 I/O automata

We describe our services and algorithms using the I/O automaton model of Lynch and Tuttle [28] (without fairness). The model and its proof methods are described in Chapter 8 of [27].

An *execution fragment* of an I/O automaton is an alternating sequence of states and actions consistent with the transition relation. An *execution* is an execution fragment that begins with a start state. The *trace* of an execution fragment α is the subsequence of α consisting of all the external actions. The external behavior of an I/O automaton is captured by the set of traces generated by its executions.

Execution fragments can be concatenated. Definitions of composition for I/O automata appear in Chapter 8 of [27], along with theorems showing that composition respects the external behavior. Invariant and simulation methods for these models are also presented in that chapter.

[1] We use "processor" and "process" interchangeably, since the difference is immaterial in the context of this paper.

3 The DVS specification

We now present DVS, our specification for a dynamic primary view group communication service.

The DVS service works as follows. Each client of the service has a "current" view of the group of processes. A process can send a message to all other members of its current view and the service guarantees that messages sent within a view are delivered only within that view and each member of the view receives messages in the same order as other members. However, not all messages need to be delivered to all members. The service also provides a "safe" notification for a particular message m that tells the recipient that message m has been received by all the members of the current view. New views are announced to all members of the new view and they are guaranteed to be "primary" views. Primary views are defined according to a dynamic notion [21]: a new primary needs to contain a majority of the members of the previous primary. The DVS service allows the clients to "register" a new view after completing the pre-processing for that view.

The specification is given in Figure 1. 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. All created views are recorded in *created*.

DVS informs its clients of view changes using DVS-NEWVIEW$(\langle g, P\rangle)_p$ actions; such an action informs processor p that the view identifier g is associated with membership 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 initial view v_0 for processors in P_0 and is undefined for other processors. 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. The variable *attempted* records, for each view, which process have been notified of that view. Variable *attempted* is only used in the proof.

With the DVS-REGISTER$_p$ action, the client at p informs the service that it has obtained whatever information the application needs to begin operating in the new view v. For many applications, this will mean that p has received messages from every other member of view v, reporting its state at the start of v. The variable *registered* records, for each view, which process have registered that view. Variable *registered* is only used in the proof.

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 other processor and without gaps in the sequence of received messages; however, a processor may receive only a prefix of the sequence of messages received by another processor. Variables *queue*, *pending*, *next* and *next-safe* are used for handling the messages. Their use should be clear from the code.

```
Signature:
Input:     $\text{DVS-GPSND}(m)_p$, $m \in \mathcal{M}_c$, $p \in \mathcal{P}$
           $\text{DVS-REGISTER}_p$, $p \in \mathcal{P}$
Internal:  $\text{DVS-CREATEVIEW}(v)$, $v \in \mathcal{V}$
           $\text{DVS-ORDER}(m,p,g)$, $m \in \mathcal{M}_c$, $p \in \mathcal{P}$, $g \in \mathcal{G}$

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

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 $\lambda$
  $attempted[g] \in 2^{\mathcal{P}}$, 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 $\lambda$
  $next[p,g] \in \mathbf{N}^{>0}$, init 1
  $next\text{-}safe[p,g] \in \mathbf{N}^{>0}$, init 1

Transitions:
  internal $\text{DVS-CREATEVIEW}(v)$
    Pre: $\forall w \in created : v.id \neq w.id$
         $\forall w \in created$ :
            $\exists x \in TotReg : w.id < x.id < v.id$
            or $\exists x \in TotReg : v.id < x.id < w.id$
            or $v.set \cap w.set \neq \{\}$
    Eff: $created := created \cup \{v\}$

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

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

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

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

  output $\text{DVS-GPRCV}(m)_{p,q}$, 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 $\text{DVS-SAFE}(m)_{p,q}$, 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 1: The DVS service

We define the following derived variables:

$\mathcal{A}tt \in 2^{\mathcal{V}}$, defined as $\{v \in \mathit{created} \mid \mathit{attempted}[v.id] \neq \{\}\}$
$\mathcal{T}ot\mathcal{A}tt \in 2^{\mathcal{V}}$, defined as $\{v \in \mathit{created} \mid v.set \subseteq \mathit{attempted}[v.id]\}$
$\mathcal{R}eg \in 2^{\mathcal{V}}$, defined as $\{v \in \mathit{created} \mid \mathit{registered}[v.id] \neq \{\}\}$
$\mathcal{T}ot\mathcal{R}eg \in 2^{\mathcal{V}}$, defined as $\{v \in \mathit{created} \mid v.set \subseteq \mathit{registered}[v.id]\}$

Informally, a view belongs to the set $\mathcal{A}tt$ if it has been reported to at least one member of the view (we say that it is *attempted*). A view belongs to the set $\mathcal{T}ot\mathcal{A}tt$ if it has been reported to all members of the view (we say that the view is *totally attempted*). Similarly, a view belongs to the set $\mathcal{R}eg$ if at least one member of the view has registered the view (we say that it is *registered*) and belongs to the set $\mathcal{T}ot\mathcal{R}eg$, if all members of the view have registered the view (we say that the view is *totally registered*).

We close this section with some invariants giving properties of DVS.

The first one is a trivial invariant which follows directly from the definition of the sets $\mathcal{A}tt, \mathcal{T}ot\mathcal{A}tt, \mathcal{R}eg$ and $\mathcal{T}ot\mathcal{R}eg$.

Invariant 3.1 (DVS)
In any reachable state, $\mathcal{T}ot\mathcal{A}tt \subseteq \mathcal{A}tt$, $\mathcal{T}ot\mathcal{R}eg \subseteq \mathcal{R}eg$, $\mathcal{R}eg \subseteq \mathcal{A}tt$, *and* $\mathcal{T}ot\mathcal{R}eg \subseteq \mathcal{T}ot\mathcal{A}tt$.

The next invariant is a basic invariant saying that if a process p has attempted a view v whose identifier is g then the current view of p is either v itself or a view with an identifier greater than g.

Invariant 3.2 (DVS)
In any reachable state if $p \in \mathit{attempted}[g]$ *then* $\mathit{current\text{-}viewid}[p] \geq g$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state $p \in \mathit{attempted}[g]$ implies that $p \in P_0$ and $g = g_0$. For $p \in P_0$ we have that $\mathit{current\text{-}viewid}[p] = g_0$ and hence the invariant is true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). The only step that changes *attempted* and *current-viewid* is $\pi =$DVS-CREATEVIEW$(v)_p$. By the precondition of π we have that for any g for which $p \in \mathit{attempted}[g]$ it holds $v.id > \mathit{current\text{-}viewid}[p]$ and by the code we have that the new value of $\mathit{current\text{-}viewid}[p]$ is $v.id$. Hence the invariant is still true. □

Invariant 3.3 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 is our version of the correctness requirement for dynamic view services that two consecutive primary views intersect.

Invariant 3.3 (DVS)
In any reachable state, if $v, w \in \mathit{created}$, $v.id < w.id$, *and there is no* $x \in \mathcal{T}ot\mathcal{R}eg$ *such that* $v.id < x.id < w.id$, *then* $v.set \cap w.set \neq \{\}$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state $\mathit{created} = \{v_0\}$ and thus the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). The only steps that can change the hypothesis from false to true are DVS-CREATEVIEW(v) and DVS-CREATEVIEW(w). The preconditions of these actions show that the needed conclusion holds. No step changes the conclusion from true to false. $\square$

Invariant 3.4 says that if a view w is totally attempted, then any earlier view v has a member whose current view is later than v.

Invariant 3.4 (DVS)
In any reachable state, if $v \in$ created, $w \in$ TotAtt, and $v.id < w.id$, then there exists $p \in v.set$ with current-viewid$[p] > v.id$.

Proof: Consider any particular reachable state. Assume that $v \in$ *created*, $w \in$ *TotAtt*, and $v.id < w.id$. Then let y be the view in *TotAtt* having the smallest viewid strictly greater than $v.id$. Then there is no $x \in$ *TotAtt* with $v.id < x.id < y.id$. Then Invariant 3.1 implies that there is no $x \in$ *TotReg* with $v.id < x.id < y.id$. Then Invariant 3.3 implies that $v.set \cap y.set \neq \{\}$. Let $p \in v.set \cap y.set$; then $p \in$ *attempted*$[y.id]$. Then Invariant 3.2 implies that *current-viewid*$[p] \geq y.id$. This implies *current-viewid*$[p] > v.id$. $\square$

4 An implementation of DVS

We now present an algorithm that implements the DVS service specification and reason about its correctness. Our implementation uses as a building block the group communication service VS [17], and it uses the ideas from [26]. The overall system is comprised of the automata VS-TO-DVS$_p$, for each $p \in \mathcal{P}$, and the VS service. We call this system DVS-IMPL and we illustrate it in Figure 2. Formally, the DVS-IMPL system is the composition of all VS-TO-DVS$_p$ automata (presented in Section 4.2) and the VS automaton (given in Section 4.1). We show that DVS-IMPL is a formal implementation of the DVS service in the sense of the *trace inclusion*, that is we prove that any trace of the DVS implementation is a trace of the DVS specification (Section 4.3).

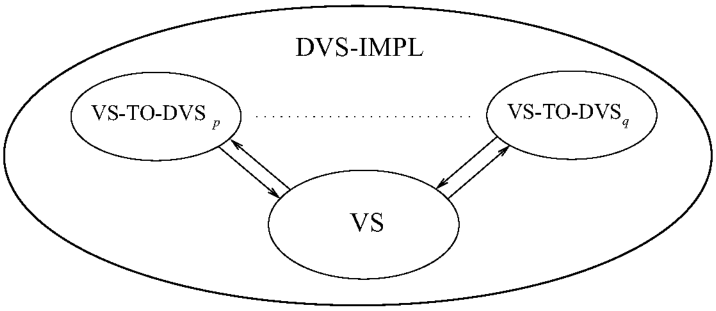

Figure 2: The DVS-IMPL system.

4.1 The VS specification

The VS service [17] is a group communication service that is similar to DVS except that VS does not provide support for primary views. The DVS service thus can be seen as an augmented version of the VS service designed to provide support for primary views. Due to the similarity of the two services, VS is a convenient building block for DVS. The specification for the VS service is given in Figure 3. To avoid a complete restatement we refer the reader to [17] for an informal description of the service.

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

output VS-SAFE$(m)_{p,q}$, choose $g, P$
  Pre: $g \neq \perp$
       $g = current\text{-}viewid[q]$
       $\langle g, P \rangle \in created$
       $queue[g](next\text{-}safe[q,g]) = \langle m,p \rangle$
       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 3: The VS service

As also reasoned in [17], the fact that VS allows views to be created only in the order of view identifier is not significant: weakening this requirement to allow out-of-order view creation does not change the external behavior, because VS-NEWVIEW actions are constrained to occur in such a way that views are always delivered in the order of view identifiers.

We rely on the following safety properties of the VS service [17]:

- New views are reported in increasing order of view identifier (monotone views property);
- Messages sent in a view are delivered only within that view (view synchrony property);
- The sequences of messages delivered in a view at any two processors are such that one sequence is a prefix of the other (prefix order property).

The following invariant holds.

Invariant 4.1 (VS)
In any reachable state, if $v, v' \in created$ and $v.id = v'.id$, then $v = v'$.

4.2 The DVS implementation algorithm and the DVS-IMPL system

The DVS implementation algorithm is given in terms of the automaton VS-TO-DVS$_p$, where $p \in \mathcal{P}$, in Figure 4. 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.

VS-TO-DVS$_p$ uses special 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 state variables *attempted*, *reg*, and *info-sent* are auxiliary – they are not needed for the algorithm, and are only used in the proofs.

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 – this is a *global* condition involving *nonempty* intersection of view sets. 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 identifiers 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 (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.

Following the DVS-NEWVIEW$_p$ even, the clients of the communication system can exchange state information as needed for processing in view v. When the 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 identifiers smaller than that of v. VS-TO-DVS uses VS to send and receive messages.

The system DVS-IMPL is defined as the composition of all the VS-TO-DVS$_p$ automata and the VS service, with all the external actions of VS hidden.

We define four derived variables for DVS-IMPL analogous to those of DVS, indicating the attempted, totally attempted, registered, and totally registered views, respectively. They are:

- $\mathcal{A}tt = \{v \in created \mid (\exists p \in v.set) v \in attempted_p\}$;

```
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 $\lambda$
    $msgs\text{-}from\text{-}vs[g] \in seqof(\mathcal{M}_c \times P)$, init $\lambda$
    $safe\text{-}from\text{-}vs[g] \in seqof(\mathcal{M}_c \times P)$, init $\lambda$
    $reg[g]$ 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 ⟨"info", $act$, $amb$⟩ to
         $msgs\text{-}to\text{-}vs[cur.id]$
       $info\text{-}sent[cur.id] := \langle act, amb \rangle$

input VS-GPRCV$(\langle \text{"info"}, v, V \rangle)_{q,p}$
  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$(\langle \text{"info"}, v, V \rangle)_{q,p}$
  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 ⟨"registered"⟩ to
           $msgs\text{-}to\text{-}vs[client\text{-}cur.id]$

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

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

internal DVS-GARBAGE-COLLECT$(v)_p$
  Pre: $\forall q \in v.set : rcvd\text{-}rgst[q, v.id] =$ 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 4: VS-TO-DVS$_p$

- $\mathcal{T}ot\mathcal{A}tt = \{v \in created \mid (\forall p \in v.set) v \in attempted_p\}$;
- $\mathcal{R}eg = \{v \in created \mid (\exists p \in v.set) reg[v.id]_p = \texttt{true}\}$; and
- $\mathcal{T}ot\mathcal{R}eg = \{v \in created \mid (\forall p \in v.set) reg[v.id]_p = \texttt{true}\}$.

Another derived variable, use_p is defined in the code of VS-TO-DVS$_p$.

4.3 Correctness of the DVS-IMPL system

We prove that DVS-IMPL implements DVS using a forward simulation argument [29] by providing an abstraction function that maps states of DVS-IMPL to states of DVS and that leads to the main observation that each trace of DVS-IMPL is a trace of DVS. We present such an abstraction function in Section 4.3.2.

Section 4.3.1 unveils a series of invariants of DVS-IMPL culminating in Invariant 4.17 and Invariant 4.18. The local condition requiring a majority intersection is captured by Invariant 4.17. Invariant 4.18 states that any two attempted views that have no intervening totally registered view have at least one member in common. This is the global condition on nonempty intersection that we have discussed in the previous section. These invariants are then used in the proof that DVS-IMPL implements DVS in Section 4.3.2.

4.3.1 Invariants

We begin with invariants that state simple facts about DVS and then proceed to more complex ones ending with the key invariant about the global condition on nonempty intersections.

Invariant 4.2 (DVS-IMPL)
In any reachable state, if $cur_p \neq \perp$ then $current\text{-}viewid[p] = cur.id_p$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p. In the initial state we have that $cur_p = \perp$.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p. We prove the invariant considering each possible action π.

1. $\pi =$ VS-NEWVIEW$(v)_p$.

   By the code of π in VS, we have that $current\text{-}viewid[p] = v.id$. By the code of π in DVS-IMPL, we have that $cur.id_p = v.id$.

2. Other actions.

   Variables $current\text{-}viewid[p]$ and $cur.id_p$ are not modified. Hence the assertion cannot be made false.

□

Invariant 4.3 (DVS-IMPL)
In any reachable state, if $v \in attempted_p$ then $client\text{-}cur.id_p \geq v.id$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix v, p. In the initial state we have that $\mathit{attempted}_p = \{v_0\}$ for $p \in P_0$ and $\mathit{attempted}_p = \bot$ for $p \notin P_0$. So assume that $v = v_0$ and $p \in P_0$. Then $\mathit{client\text{-}cur}_p = v_0$. Hence the invariant is true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix v, p and assume that $v \in s'.\mathit{attempted}_p$. We distinguish two possible cases.

1. $v \in s.\mathit{attempted}_p$.

   By the inductive hypothesis we have that $s.\mathit{client\text{-}cur}_p \geq v.id$. By the monotonicity of $\mathit{client\text{-}cur}_p$ we have that $s'.\mathit{client\text{-}cur}_p \geq s.\mathit{client\text{-}cur}_p$.

2. $v \notin s.\mathit{attempted}_p$.

   Then it must be $\pi =$DVS-NEWVIEW$(v)_p$. The invariant follows from the code which sets $\mathit{client\text{-}cur}_p$ to v.

□

Invariant 4.4 (DVS-IMPL)
In any reachable state, if $v \in \mathit{info\text{-}sent}[g]_p = \langle x, X \rangle$ *then* $cur.id_p \geq g$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix v, p. In the initial state we have that $\mathit{info\text{-}sent}_p = \bot$ and thus the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, x, X and assume that $s'.\mathit{info\text{-}sent}[g]_p = \langle x, X \rangle$. We distinguish two possible cases.

1. $s.\mathit{info\text{-}sent}[g]_p = \langle x, X \rangle$

   By the inductive hypothesis we have that $s.cur_p \geq g$. By the monotonicity of cur_p we have that $s'.cur_p \geq s.cur_p$. Hence the invariant is true.

2. $s.\mathit{info\text{-}sent}[g]_p \neq \langle x, X \rangle$

   Then it must be $\pi =$VS-NEWVIEW$(v)_p$ and $g = v.id = s'.act.id_p$. Action VS-NEWVIEW$(v)_p$ sets $s'.cur$ to v, so $s'.cur.id = g$.

□

Invariant 4.5 (DVS-IMPL)
In any reachable state:

1. $v_0 \in \mathcal{TotReg}$.

2. $g_0 \leq v.id$ *for all* $v \in$ *created.*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Part 1 is true because in then initial state every processor $p \in P_0$ has $reg[g_0] = \texttt{true}$. Part 2 is true because the only view in *created* is v_0.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s').

Consider Part 1 first. No view is ever removed from *TotReg*. Hence no step can make the assertion false. Consider Part 2 now. Fix v and assume that $v \in s'.created$. We distinguish two cases.

1. $v \in s.created$.

   Then the assertion follows from the inductive hypothesis.

2. $v \notin s.created$.

   It must be π=VS-CREATEVIEW$(v)_p$. By the precondition of this action we have that $v.id > w.id$ for all $w \in s.created$. By the inductive hypothesis $g_0 \leq w.id$ for all $w \in s.created$. Since $s'.created = s.created \cup \{v\}$, it follows that $g_0 \leq w.id$ for all $w \in s'.created$.

□

Invariant 4.6 (DVS-IMPL)
In any reachable state, if $rcvd\text{-}rgst[q, v.id]_p \neq \bot$ *then* $cur_p \neq \bot$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, q and v. In the initial state we have that $rcvd\text{-}rgst[q, v.id]_p = \bot$. Hence the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, q, v. We prove the invariant considering each possible action π. Assume that $s'.rcvd\text{-}rgst[q, v.id]_p \neq \bot$.

1. π = VS-NEWVIEW$(v)_p$.

   Since $s'.cur_p = v$ we have that $s'.cur_p \neq \bot$ (VS cannot deliver $\bot$, it is not a view).

2. π = VS-GPRCV$(\langle \text{“}registered\text{”}\rangle)_{p,q}$.

   By the precondition of π (see VS) we have that $s.current\text{-}viewid[p] \neq \bot$. By Invariant 4.2 we have $s.cur.id_p = s.current\text{-}viewid[p] \neq \bot$. Hence $s'.cur.id_p = s.cur.id_p \neq \bot$.

3. Other actions.

   Variables $rcvd\text{-}rgst[q, v.id]_p$ and cur_p are not modified. Hence the assertion cannot be made false.

□

Invariant 4.7 (DVS-IMPL)
In any reachable state, if $cur.id_p = \bot$ *then* $act_p = v_0$ *and* $amb_p = \{\}$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p. In the initial state we have that $act_p = v_0$ and $amb_p = \{\}$.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p. We prove the invariant considering each possible action π. Assume that $s'.cur_p = \perp$. Since no actions sets cur_p to $\perp$ it must be $s.cur_p = \perp$.

1. $\pi = \text{VS-GPRCV}(\langle \text{"}info\text{"}, v, V\rangle)_{p,q}$.

   This cannot happen. Indeed by precondition of π (see VS) we have that $s.current\text{-}viewid[p] \neq \perp$. By Invariant 4.2 we have $s.cur.id_p = s.\text{VS}.current\text{-}viewid[p]$ Hence $s'.cur.id_p = s.cur.id_p \neq \perp$. But we know that $s'.cur.id = \perp$.

2. $\pi = \text{DVS-NEWVIEW}(v)v$.

   Cannot happen. Indeed the precondition of π says that $v = s.cur_p$. Since $s.cur.id = \perp$, we have $v = \perp$. Thus the precondition $v.id > client\text{-}cur.id_p$ cannot be satisfied ($\perp$ cannot be strictly greater than any view identifier).

3. $\pi = \text{DVS-GARBAGE-COLLECT}(v)$.

   Cannot happen. Indeed by Invariant 4.6 we have that $s.cur_p \neq \perp$. But we know that $s.cur_p = \perp$.

4. Other actions.

   Variables cur_p, act_p and amb_p are not modified. Hence the assertion cannot be made false.

□

The following invariant states that if an *"info"* message is in transit for view v or has been received by some process q in view v then there exists a process p that has sent the *"info"* in view v and such that its current view is either v or a later one.

Invariant 4.8 (DVS-IMPL)
In any reachable state, let C be the following condition:

> $\langle \text{"}info\text{"}, x, X\rangle \in msgs\text{-}to\text{-}vs[g]_p$ *or* $\langle \text{"}info\text{"}, x, X\rangle \in pending[p, g]$ *or* $\langle\langle \text{"}info\text{"}, x, X\rangle, p\rangle \in queue[g]$ *or* $info\text{-}rcvd[p, g]_q = \langle x, X\rangle$.

If C is true then $info\text{-}sent[g]_p = \langle x, X\rangle$ and $cur.id_p \geq g$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, q, g, x and X. In the initial state $msgs\text{-}to\text{-}vs[g]_p = \lambda$, $pending[p, g] = \lambda$, $queue[g] = \lambda$ and $info\text{-}rcvd[p, g]_q = \perp$. Hence, in the initial state, C is false and the invariant is vacuously true.

For the inductive step assume that the invariant is true in a reachable state s. We need to prove that it is true in state s' for any possible step (s, π, s') of the execution. Fix p, q, g, x, and X and assume that C is true in s'.

1. $\pi = \text{VS-NEWVIEW}(v)_p$.

   By the code of π, $s'.cur_p = v$. Assume $v.id \neq g$. Then the code of π shows that none of $msgs\text{-}to\text{-}vs[g]_p$, $pending[p,g]$, $queue[g]$ or $info\text{-}rcvd[p,g]_q$ is changed during this step. Thus C is true also in s. By the inductive hypothesis we have $s.info\text{-}sent[g]_p = \langle x, X\rangle$ and $cur.id_p \geq g$. Since we are considering the case $v.id \neq g$, we have that $info\text{-}sent[g]_p$ is not changed by π. Moreover the precondition of π (see VS) shows that $s'.current\text{-}viewid[p] > s.current\text{-}viewid[p]$. By Invariant 4.2, $cur.id_p = current\text{-}viewid[p]$, so $s'.cur.id_p > s.cur.id_p$. This completes showing the conclusion for the situation $w.id \neq g$.

   Assume now $v.id = g$. The code shows $s'.cur.id_p = g$ as required. It remains to show that $\langle x, X\rangle \in info\text{-}sent[g]_p$.
   Action π does not alter the values of $pending[p,g]$, $queue[g]$ and $info\text{-}rcvd[p,g]_q$ and appends $\langle \text{"info"}, s.act_p, s.amb_p\rangle$ to $msgs\text{-}to\text{-}vs[g]_p$. We claim that it must be $x = s.act_p$ and $X = s.amb_p$. Indeed if it is not so, then condition C is true also in state s (for the given p, q, g, x, X) and by the inductive hypothesis we have $s.cur.id_p \geq g = w.id$. By Invariant 4.2, $s.current\text{-}viewid[p] \geq w.id$. But this contradicts the precondition of π (see VS).
   Thus $x = s.act_p$ and $X = s.amb_p$. Then the code of π shows that $\langle x, X\rangle \in info\text{-}sent[g]_p$, as required.

2. $\pi = \text{VS-GPRCV}(\langle \text{"info"}, v, V\rangle)_{p,q}$.

   If $g \neq cur.id_q$ then since C is true in s' it is true also in s (for the given p, q, g, x, X). Thus the inductive hypothesis is true. Since the code does not change $info\text{-}sent[g]_p$ and $cur.id_p$, the invariant follows from the inductive hypothesis.

   Hence assume that $g = cur.id_q$. First consider the case $x = v$ and $X = V$. In this case, by the precondition of π (see VS) we have that $\langle\langle \text{"info"}, x, X\rangle, p\rangle \in queue[g]$. Then the invariant follows from the inductive hypothesis.

   Consider now the case $x \neq v$ or $X \neq V$. In this case, by the code, we have that $s'.info\text{-}rcvd[p,g]_q \neq \langle x, X\rangle$. Since C is true in s', it must be that $\langle \text{"info"}, x, X\rangle \in msgs\text{-}to\text{-}vs[g]_p$ or $\langle \text{"info"}, x, X\rangle \in pending[p,g]$ or $\langle\langle \text{"info"}, x, X\rangle, p\rangle \in queue[g]$ is true in s'. Variables $msgs\text{-}to\text{-}vs[g]_p$, $pending[p,g]$ and $queue[g]$ are not changed by π. Hence C is true in s. The invariant follows from the inductive hypothesis.

3. $\pi = \text{VS-GPSND}(\langle \text{"info"}, v, V\rangle)_p$.

   If $g \neq client\text{-}cur.id_p$ then since C is true in s' it is true also in s (for the given p, q, g, x, X). Thus the inductive hypothesis is true. Since the code does not change $info\text{-}sent[g]_p$ and $cur.id_p$, the invariant follows from the inductive hypothesis.

   Hence assume that $g = client\text{-}cur.id_p$. First consider the case $x = v$ and $X = V$. In this case, by the precondition of π (see DVS-IMPL) we have that $\langle\langle \text{"info"}, x, X\rangle, p\rangle \in msgs\text{-}to\text{-}vs[g]$. Then the invariant follows from the inductive hypothesis.

   Consider now the case $x \neq v$ or $X \neq V$. Since C is true in s' we have that C is true in s too. Indeed no $\langle \text{"info"}, x, X\rangle$ message is deleted and $info\text{-}rcvd[p,g]_q$ is not changed. The invariant follows from the inductive hypothesis.

4. $\pi = $ VS-ORDER$(\langle$ *"info"*$, v, V\rangle, p, g)$.

   First consider the case $x = v$ and $X = V$. In this case, by the precondition of π we have that $\langle\langle$ *"info"*$, x, X\rangle, p\rangle \in$ *pending*$[g]$. Then the invariant follows from the inductive hypothesis.

   Consider now the case $x \neq v$ or $X \neq V$. Since C is true in s' we have that C is true in s too. Indeed no $\langle$ *"info"*$, x, X\rangle$ message is deleted and *info-rcvd*$[p, g]_q$ is not changed. The invariant follows from the inductive hypothesis.

5. Other actions.

   Condition C never changes from false to true and variables *info-sent*$[g]_p$ and *cur.id*$_p$ are not modified. Hence the assertion cannot be made false.

□

The following invariant states that if a *"registered"* message for view v has been sent by process p then variable *reg*$[v.id]_p$ is set to true (that is, the view has been registered by the client at p).

Invariant 4.9 (DVS-IMPL)
In any reachable state, let C be the following condition:

$\langle$ *"registered"*$\rangle \in$ *msgs-to-vs[g]*$_p$ *or* $\langle$ *"registered"*$\rangle \in$ *pending*$[p, g]$ *or* $\langle$ *"registered"*$, p\rangle \in$ *queue*$[g]$ *or* *rcvd-rgst*$[p, g]_q = \texttt{true}$.

If C is true then *reg*$[g]_p = \texttt{true}$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, g, q. In the initial state we have that *msgs-to-vs[g]*$_p = \lambda$, *pending*$[p, g] = \lambda$, *queue*$[g] = \lambda$ and *rcvd-rgst*$[p, g]_q = \texttt{false}$. Hence C is false in the initial state and the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, q and assume that C is true in s'.

1. $\pi = $ DVS-REGISTER$_p$.

   If s.*client-cur.id*$_p \neq g$ then C is true also in s and the invariant follows from the inductive hypothesis. Hence assume s.*client-cur.id*$_p = g$. By the code of π we have that we have *reg*$[g]_p = \texttt{true}$.

2. $\pi = $ VS-GPSND$(\langle$ *"registered"*$\rangle)_p$.

   If s.*current-viewid*$[p] \neq g$ then C is true also in s and the invariant follows from the inductive hypothesis. Hence assume $g = s$.*current-viewid*$[p]$. By Invariant 4.2 we have that s.*cur.id*$_p = s$.*current-viewid*$[p]$. Hence s.*cur.id*$_p = g$. By the precondition of π (see DVS-IMPL) we have that $\langle$ *"registered"*$\rangle \in s$.*msgs-to-vs*$[g]_p$. Hence C is true in s and the invariant follows from the inductive hypothesis.

3. $\pi =$VS-ORDER$(\langle$ *"registered"*$, p', g'\rangle)$.

   If $p' \neq p$ or $g' \neq g$ then C is true also in s and the invariant follows from the inductive hypothesis. Hence assume $p' = p$ and $g' = g$. By the precondition of π we have that $\langle$ *"registered"*$\rangle \in s$.*pending*$[p, g]$. Hence C is true also in s and the invariant follows from the inductive hypothesis.

4. Other actions.

   Condition C never changes from false to true and variable $reg[g]_p$ is not modified. Hence the assertion cannot be made false.

□

The following invariant states some facts about views in TotReg.

Invariant 4.10 (DVS-IMPL)
In any reachable state:

1. $act_p \in \mathit{TotReg}$.

2. If $info\text{-}sent[g]_p = \langle x, X\rangle$ *then* $x \in \mathit{TotReg}$.

3. $use_p \cap \mathit{TotReg} \neq \{\}$.

Proof: First notice that Part 3 follows easily from Part 1 and the fact that, by definition, $act_p \in use_p$. Hence we only need to prove Parts 1 and 2.

By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. For Part 1, fix p. In the initial state $act_p = v_0$ and v_0 is totally registered by definition. For Part 2, fix p, g. In the initial state $info\text{-}sent[g]_p = \perp$. Hence the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, x and X. We prove the invariant by considering each possible action.

1. $\pi =$ VS-NEWVIEW$(v)_p$.

   Part 1 is still true in s' because act_p is not modified (as well as TotReg).

   Consider Part 2 now. Assume that $s'.info\text{-}sent[g]_p = \langle x, X\rangle$. If $v.id \neq g$ then $s.info\text{-}sent[g]_p = \langle x, X\rangle$ then by the inductive hypothesis we have that $x \in s.\mathit{TotReg}$. Since no view is ever removed from TotReg we have that $x \in s'.\mathit{TotReg}$, as needed. Hence we can further assume that $v.id = g$. Since $s'.info\text{-}sent[g]_p = \langle x, X\rangle$ and action π sets $info\text{-}sent[g]_p = \langle act_p, amb_p\rangle$ it must be that $s.act_p = x$ and $s.amb_p = X$.

   By the inductive hypothesis, Part 1, we have that $s.act_p \in s.\mathit{TotReg}$. But $x = s.act_p$ and no view is removed from TotReg. Hence $x \in s'.\mathit{TotReg}$. Thus Part 2 is still true in s'.

2. $\pi =$ VS-GPRCV$(\langle \text{“}info\text{”}, v, V\rangle)_{p,q}$.

   Consider Part 1 first. If $s'.act_p = s.act_p$ then Part 1 follows by the inductive hypothesis. Hence assume that $s'.act_p \neq s.act_p$. By the code we have that $s'.act_p = v$. Thus we have to prove that $v \in \mathit{TotReg}$. By the precondition of π (in VS) we have $\langle\langle \text{“}info\text{”}, v, V\rangle, q\rangle \in s.queue[cur.id_p]$. Then Invariant 4.8 implies that $s.info\text{-}sent[cur.id_p]_q = \langle v, V\rangle$. By the inductive hypothesis, Part 2, we have that $v \in s.\mathit{TotReg}$, as needed.

   Part 2 is preserved because $info\text{-}sent[g]_p$ is not modified.

3. $\pi = \text{DVS-GARBAGE-COLLECT}(v)_p$.

   Consider Part 1 first. If $s'.act_p = s.act_p$ then Part 1 follows by the inductive hypothesis. Hence assume that $s'.act_p \neq s.act_p$. By the code we have that $s'.act_p = v$. Hence we have to prove that $v \in \mathit{TotReg}$. By the precondition of π we have that $\mathit{rcvd\text{-}rgst}[q, v.id] = \texttt{true}$ for all $q \in v.set$. Then Invariant 4.9 implies that $v \in \mathit{TotReg}$.

   Part 2 is preserved because $\mathit{info\text{-}sent}[g]_p$ is not modified.

4. Other actions.

   Variables act_p, $\mathit{info\text{-}sent}[g]_p$ (as well as TotReg) are not modified. Hence the assertions cannot be made false.

□

The following invariant states that if process q is in a view which has been attempted by process p (which may or may not be q itself) then the current view of q is either v or a later one.

Invariant 4.11 (DVS-IMPL)
In any reachable state, if $v \in attempted_p$ and $q \in v.set$ then $cur.id_q \geq v.id$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, v and suppose that $v \in attempted_p$ and $q \in v.set$. If $p \notin P_0$ then $attempted_p = \{\}$, a contradiction. On the other hand, if $p \in P_0$ then since $v \in attempted_p$, it must be that $v = v_0$. Moreover since $q \in v.set$ we have that $q \in P_0$. Hence $cur_q = v_0$, so $cur.id_q \geq v.id$, as needed.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p and v and assume that $v \in s'.attempted_p$ and $q \in v.set$. We distinguish two cases.

1. $v \in s.attempted_p$.

   By the inductive hypothesis we have that $s.cur.id_q \geq v.id$. By the monotonicity of $cur.id$ we have that $s'.cur.id_q \geq s.cur.id_q$.

2. $v \notin s.attempted_p$.

   It must be $\pi = \text{DVS-NEWVIEW}(v)_p$. We consider two possible cases: $q = p$ and $q \neq p$.

   Assume that $q = p$. Then Invariant 4.3 implies that $s'.\mathit{client\text{-}cur}_p \geq v.id$. Since $s'.cur.id_p = s'.\mathit{client\text{-}cur}_p$, we have that $s'.cur.id_p \geq v.id$, as needed.

   Assume that $q \neq p$. Then the precondition of π says that $s.\mathit{info\text{-}rcvd}[q, v.id] \neq \bot$. By Invariant 4.8 (used with p and q interchanged) we have that $cur.id_q \geq v.id$, as needed.

□

The following invariant states properties of views in the *use* set.

Invariant 4.12 (DVS-IMPL)
In any reachable state:

1. If $cur_p \neq \bot$ and $w \in use_p$, then $w.id \leq cur.id_p$.

2. If $cur_p \neq \bot$ and $client\text{-}cur_p \neq cur_p$ and $w \in use_p$, then $w.id < cur.id_p$.

3. If $info\text{-}sent[g]_p = \langle x, X \rangle$ and $w \in \{x\} \cup X$ then $w.id < g$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Consider Part 1 first. In the initial state we have that use_p is either empty or contains only v_0. In the former case Part 1 is vacuously true. In the latter case we have that $w = v_0$ and the invariant follows from the fact that g_0 is the minimum element of $\mathcal{G}$. Parts 2 and 3 are vacuously true. Indeed in the initial state $client\text{-}cur_p = cur_p$ and $info\text{-}sent[g]_p = \bot$.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, x, X and w.

We prove that the invariant is still true in s' by considering each possible action π.

1. $\pi =$ VS-NEWVIEW$(v)_p$

   First consider Part 1. Assume that $s'.cur_p \neq \bot$ and $w \in s'.use_p$. Then $w \in s.use_p$. If $s.cur_p = \bot$, then, by Invariant 4.7, $w = v_0$. Since $v_0.id$ is the minimum element of $\mathcal{G}$, we have that $w.id < s'.cur.id_p$. So assume that $s.cur_p \neq \bot$. In this case, by the inductive hypothesis, Part 1, we have that $w.id \leq s.cur.id_p$, which implies $w.id < s'.cur.id_p$.

   Hence Part 1 is still true in s'. Since we actually proved that $w.id < s'.cur.id_p$ also Part 2 is still true in s'.

   Now consider Part 3. Assume that $s'.info\text{-}sent[g]_p = \langle x, X \rangle$ and $w \in \{x\} \cup X$. If $g \neq v.id$ then we have that $s.info\text{-}sent[g]_p = \langle x, X \rangle$. By the inductive hypothesis, Part 3, we have $w.id < g$, as needed. Hence assume $g = v.id$. By the code of π, we have that $s.use_p = \{x\} \cup X$. Now if $s.cur_p = \bot$, then by Invariant 4.7, $w = v_0$. Since $v_0.id$ is the minimum element of $\mathcal{G}$, we have that $w.id < v.id = g$, as needed. So assume further that $s.cur_p \neq \bot$. In this case, the inductive hypothesis, Part 1, implies that $w.id \leq s.cur.id_p$, which implies $w.id < s'.cur.id_p = v.id = g$, as needed.

2. $\pi =$ DVS-NEWVIEW$(v)_p$

   Consider Part 1 first. The only possible new element added to use_p is v. Since $v = s'.cur.id$, Part 1 still holds in s'. Part 2 is vacuously true, because $s'.client\text{-}cur_p = s'.cur_p$. Part 3 is preserved because $info\text{-}sent[g]_p$ is not modified.

3. $\pi =$ DVS-GARBAGE-COLLECT$(v)_p$

   Consider Part 1. Assume that $s'.cur_p \neq \bot$ and that $w \in s'.use_p$. By the code $s'.cur_p = s.cur_p$. If $w \in s.use_p$ then by the inductive hypothesis Part 1 is true in s and thus it is still true in s'. Hence assume that $w \notin s.use_p$. By the code, this cannot happen because no view is added to use_p.

   Part 2 can be proved in a similar way. Part 3 is preserved because $info\text{-}sent[g]_p$ is not modified.

4. $\pi =$ VS-GPRCV$(\langle \text{“}info\text{”}, x, X \rangle)_{q,p}$

   The proof is exactly as in the previous case.

5. Other actions.

   Variables use_p, cur_p, $client\text{-}cur_p$ and $info\text{-}sent[g]_p$ are not modified. Hence none of the assertions can be made false.

□

The following three invariants, say that certain views appear in *use* sets, or in *"info"* messages, unless they have been garbage-collected.

Invariant 4.13 (DVS-IMPL)
In any reachable state, if $w \in attempted_p$ then either $w \in use_p$ or $w.id < act.id_p$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, w and suppose that $w \in attempted_p$. If $p \notin P_0$ then $attempted_p = \{\}$, a contradiction. On the other hand, if $p \in P_0$ then since $w \in attempted_p$, it must be that $w = v_0$. But in this case also $act_p = v_0$, so $v_0 \in use_p$, as needed.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). So fix w and p such that $w \in s'.attempted_p$. We distinguish two possible cases.

1. $w \in s.attempted_p$.

   By the inductive hypothesis we have that either $w \in s.use_p$ or $w.id < s.act.id_p$. In the latter case, because of the monotonicity of $act.id_p$, we have $w.id < s'.act.id_p$. So assume that $w \in s.use_p$. If $w \in s'.use_p$ we are done, so assume further that $w \notin s'.use_p$. Then it must be that either $\pi =$ DVS-GARBAGE-COLLECT$(v)_p$ or $\pi =$ VS-GPRCV$(\langle \text{"}info\text{"}, x, X\rangle)_{r,p}$ for some r. In either case, the code implies that $s'.act_p > w.id$.

2. $w \notin s.attempted_p$.

   It must be $\pi =$ DVS-NEWVIEW$(v)_p$. By the code, view v is inserted into $attempted_p$, but also into amb_p (and hence into use_p). Thus the invariant is still true in s'.

□

Invariant 4.14 (DVS-IMPL)
In any reachable state, if $info\text{-}rcvd[q,g]_p = \langle x, X\rangle$ and $w \in \{x\} \cup X$, then either $w \in use_p$ or $w.id < act.id_p$.

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state $info\text{-}rcvd[q,g]_p = \perp$ for any p, q, g. Hence the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, q, g, x, X and w, and assume that $s'.info\text{-}rcvd[q,g]_p = \langle x, X\rangle$, and $w \in \{x\} \cup X$. We consider two cases:

1. $s.info\text{-}rcvd[q,g]_p = \langle x, X\rangle$

   By the statement applied to s, we obtain that either $w \in s.use_p$, or $s.act.id_p > w.id$. In the latter case, $s'.act.id_p > w.id$, because of monotonicity of $act.id_p$. So assume that $w \in s.use_p$.

If $w \in s'.use_p$ then we are done, so assume further that $w \notin s'.use_p$. (That is, w is garbage-collected.)
Then it must be that either π =DVS-GARBAGE-COLLECT$(v)_p$ or π = VS-GPRCV$(\langle \text{"}info\text{"}, x, X\rangle)_{r,p}$ for some r. In either case, the code implies that $s'.act_p > w.id$.

2. $s.info\text{-}rcvd[q, g]_p \neq \langle x, X\rangle$

   Then π = VS-GPRCV$(\langle \text{"}info\text{"}, x, X\rangle)_{q,p}$. If $w \in s'.use_p$ then we are done. Hence assume that $w \notin s'.use_p$. By the code, we have that $s'.act_p > w.id$ (that is, w is garbage-collected).

□

Invariant 4.15 (DVS-IMPL)
In any reachable state, if info-sent$[g]_p = \langle x, X\rangle$*,* $w \in attempted_p$*, and* $w.id < g$*, then either* $w \in \{x\} \cup X$ *or* $w.id < x.id$*.*

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, $info\text{-}sent[g]_p = \perp$ for all g, p, so the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, w, x, and X, and assume that $s'.info\text{-}sent[g]_p = \langle x, X\rangle$, $w \in s'.attempted_p$, and $w.id < g$. We consider four cases:

1. $s.info\text{-}sent[g]_p = \langle x, X\rangle$ and $w \in s.attempted_p$.

   Then the statement for s implies that either $w \in \{x\} \cup X$ or $w.id < x.id$. In either case the statement is true in s' also.

2. $s.info\text{-}sent[g]_p \neq \langle x, X\rangle$ and $w \notin s.attempted_p$.

   This cannot happen because both conditions cannot become true in a single step: the first only becomes true by means of a VS-NEWVIEW$(v)_p$, for some view v, while the second only becomes true by means of DVS-NEWVIEW$(w)_p$.

3. $s.info\text{-}sent[g]_p \neq \langle x, X\rangle$ and $w \in s.attempted_p$.

   It must be π = VS-NEWVIEW$(v)_p$, for some v, x must be $s.act_p$, and X must be $s.amb_p$. Invariant 4.13 implies that either $w \in s.use_p$ or $w.id < s.act.id_p$. Now, $s.use_p = \{s.act_p\} \cup s.amb_p = \{x\} \cup X$. So we have that either $w \in \{x\} \cup X$ or $w.id < x.id$, as needed.

4. $s.info\text{-}sent[g]_p = \langle x, X\rangle$ and $w \notin s.attempted_p$.

   Then π must be DVS-NEWVIEW$(w)_p$. We claim that this cannot happen: Since $s.info\text{-}sent[g]_p = \langle x, X\rangle$, by Invariant 4.4 we have $s.cur.id_p \geq g$. Since $g > w.id$, we have $s.cur_p > w.id$. But the precondition of π requires that $s.cur_p = w.id$. Hence π is not enabled in state s.

□

Invariant 4.16 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 processor that attempts the second view.

Invariant 4.16 (DVS-IMPL)
In any reachable state, 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. The base case consists of proving that the invariant is true in the initial state. In the initial state, only v_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). 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$(v)_p$. Since $q \in v.set$, 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. Then Invariant 4.8 implies that $s.info\text{-}sent[v.id]_q = \langle x, X\rangle$. Then (since $w.id < v.id$), Invariant 4.15 implies that either $w \in \{x\} \cup X$ or $w.id < x.id$. If $w.id < x.id$, then we obtain a contradiction. Indeed by Invariant 4.10 $x \in s.\mathcal{T}ot\mathcal{R}eg$ and by Invariant 4.12, Part 3 (used with $w = x$) we have $x.id < v.id$. This contradicts the hypothesis. So $w \in \{x\} \cup X$.
   Now by Invariant 4.14 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. Indeed by Invariant 4.10 we have $s.act_p \in \mathcal{T}ot\mathcal{R}eg$. Moreover by the precondition of π, $s.cur_p$ cannot be $\perp$ and $s.cur_p > s.client\text{-}cur_p$ and, by definition, $s.act_p \in s.use_p$. Hence by Invariant 4.12, Part 2, we have $s.act.id_p < s.cur_p = v.id$.

4. $v \in s.attempted_p$ and $w \notin s.attempted_q$.

   Then π must be DVS-NEWVIEW$(w)_q$. But this cannot happen. Indeed since $v \in s.attempted_p$ and $q \in v.set$, Invariant 4.11 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 4.17 says that any attempted view v intersects the latest preceding totally registered view w in a majority of members of w.

Invariant 4.17 (DVS-IMPL)
In any reachable state, 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. The base case consists of proving that the invariant is true in the initial state. In the initial state, only v_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). 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 $x \in s'.\mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. We consider four cases:

1. $v \in s.\mathcal{A}tt$ and $w \in s.\mathcal{T}ot\mathcal{R}eg$.

   Then, from the inductive hypothesis we have $|v.set \cap w.set| > |w.set|/2$.

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 4.10, Part 3, $s.use_p \cap s.\mathcal{T}ot\mathcal{R}eg \neq \{\}$. Let m be the view in $s.use_p \cap s.\mathcal{T}ot\mathcal{R}eg$ having the biggest identifier. We know that $m \neq w$ because $w \notin s.use_p$. Also, $m \neq v$, because $m \in s.\mathcal{T}ot\mathcal{R}eg$ and $v \notin s.\mathcal{T}ot\mathcal{R}eg$. It follows that $m.id \neq v.id$.

   We claim that $m.id < w.id$. We have already shown that $m.id \neq w.id$. Suppose for the sake of contradiction that $m.id > w.id$. From the precondition of action π we have that $s.cur = v$ and hence $s.cur \neq \perp$. Also from the precondition of π we have that $s.client\text{-}cur_p < s.cur_p$. Since $m \in s.use_p$, Invariant 4.12, Part 2, implies that $m.id < s.cur.id_p$ and since $s.cur = v$ we have we have $m.id < v.id$. 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. Remember that $w \in s'.\mathcal{T}ot\mathcal{R}eg$ and since π =DVS-NEWVIEW$(v)_p$ we have that $w \in s.\mathcal{T}ot\mathcal{R}eg$; thus n exists and it holds $m.id < n.id \leq w.id < v.id$. 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. Then Invariant 4.8 implies that $s.info\text{-}sent[v.id]_q = \langle x, X\rangle$. Then Invariant 4.12, Part 3 (used with $w = x$), 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 4.15 (used with $w = n$) implies that either $n \in \{x\} \cup X$ or $n.id < x.id$. In either case, $\{x\} \cup X$ contains a view $y \in s.\mathcal{T}ot\mathcal{R}eg$ (either n or x) such that $n.id \leq y.id < v.id$. Then Invariant 4.14 implies that either $y \in s.use_p$ or $y.id < s.act.id_p$. By Invariant 4.10, Part 1, $s.act_p \in s.\mathcal{T}ot\mathcal{R}eg$ 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 4.16 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 4.17, is instrumental in proving that DVS-IMPL implements DVS.

Invariant 4.18 (DVS-IMPL)
In any reachable state, 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. $w \in \mathcal{T}ot\mathcal{R}eg$.

   Since there is no $x \in \mathcal{T}ot\mathcal{R}eg$, Invariant 4.17 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 $Y = \{y | y \in \mathcal{T}ot\mathcal{R}eg, y.id < w.id\}$. We first show that Y is nonempty: Invariant 4.5 implies 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 Invariant 4.1, we have $w = v_0$. But then $w \in \mathcal{T}ot\mathcal{R}eg$, a contradiction to the definition of this case. 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. We have that there is no $x \in \mathcal{T}ot\mathcal{R}eg$ with $z.id < x.id < v.id$. Then Invariant 4.17 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.

□

4.3.2 The abstraction function

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 *abstraction function*. Section 4.3.2.1 contains the definition of the function $\mathcal{F}$ along with auxiliary invariants, then Section 4.3.2.2 gives the proof that $\mathcal{F}$ is an abstraction function.

4.3.2.1 Definition of $\mathcal{F}$

DVS-IMPL uses VS to send client messages and messages generated by the implementation (*"info"* and *"registered"* messages). The abstraction function 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 5 defines the abstraction function $\mathcal{F}$.

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 $g \in \mathcal{G}$, $t.attempted[g] = \{p | g = v.id, v \in s.attempted_p\}$
- 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 5: The abstraction function $\mathcal{F}$.

Next we give some simple consequences of the definition of $\mathcal{F}$. They deal with the messages delivered by DVS-IMPL. They state that these messages are exactly the ones that DVS would deliver to the client.

Invariant 4.19 (DVS-IMPL)
In any reachable state s, if $s.msgs\text{-}from\text{-}vs[g]_p = \langle\langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$, then we have that $\mathcal{F}(s).queue[g](next[p, g]..next[p, g] + k - 1) = \langle\langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state no message is in $msgs\text{-}from\text{-}vs[g]_p$. Hence the invariant is vacuously true.
For the inductive step, assume that the invariant is true in state s. We need to prove that it is true in state s' for any possible step (s, π, s'). Fix p, g and $m_1, q_1, m_2, q_2, ..., m_k, q_k$ and assume that $s'.msgs\text{-}from\text{-}vs[g]_p = \langle\langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$. We distinguish the following cases.

1. $s.msgs\text{-}from\text{-}vs[g]_p = \langle\langle m_1, q_1\rangle, ..., \langle m_{k-1}, q_{k-1}\rangle\rangle$.

   It must be $\pi = \text{VS-GPRCV}(m_k)_{q_k,p}$. By the inductive hypothesis we have that
   $\mathcal{F}(s).queue[g](next[p, g]..next[p, g] + k - 2) = \langle\langle m_1, q_1\rangle, ..., \langle m_{k-1}, q_{k-1}\rangle\rangle$.

By the code in VS we have that $next[p,g]$ is increased by one and by the code in DVS we have that the size of $msgs\text{-}from\text{-}vs[g]_p$ also increases by one. Hence by the definition of F, we have that $\mathcal{F}(s').next[p,g] = \mathcal{F}(s).next[p,g]$. Moreover $\mathcal{F}(s').queue[g] = \mathcal{F}(s).queue[g]$ and by the precondition of π we have that $\mathcal{F}(s).queue[g](s.next[p,g] + k - 1) = \langle m_k, q_k\rangle$. Thus the invariant is still true in s'.

2. $s.msgs\text{-}from\text{-}vs[g]_p = \langle\langle m,q\rangle, \langle m_1,q_1\rangle, \langle m_2,q_2\rangle, ..., \langle m_k,q_k\rangle\rangle$.

   Then $\pi =$DVS-GPRCV$(m)_{q,p}$. By the inductive hypothesis we have that $\mathcal{F}(s).queue[g](next[p,g]..next[p,g] + k) = \langle\langle m,q\rangle, \langle m_1,q_1\rangle, \langle m_2,q_2\rangle, ..., \langle m_{k-1},q_{k-1}\rangle\rangle$.

   By the code we have that $next[p,g]$ is incremented by one. Since $\mathcal{F}(s').queue[g] = \mathcal{F}(s).queue[g]$, the invariant is still true in s'.

3. $s.msgs\text{-}from\text{-}vs[g]_p = s'.msgs\text{-}from\text{-}vs[g]_p$

   By the inductive hypothesis the assertion is true in state s. For any possible action in this case $\mathcal{F}(s').next[p,g] = \mathcal{F}(s).next[p,g]$ and the portion of $\mathcal{F}(s).queue[g]$ involved in the statement of the invariant never changes because messages are only appended to $queue[g]$. Thus the assertion cannot be made false.

4. Other cases.

   Not possible. Indeed $msgs\text{-}from\text{-}vs[g]_p$ either stay the same or is changed by appending a message or deleting the head.

□

The following invariant follows easily from the previous one. It just states that the next message delivered by DVS-IMPL to a processor p is the same one that DVS delivers.

Invariant 4.20 (DVS-IMPL)
In any reachable state s, if $\langle m,q\rangle$ is head of $s.msgs\text{-}from\text{-}vs[g]_p$, then $\mathcal{F}(s).queue[g](next[p,g]) = \langle m,q\rangle$.

Proof: Follows easily from previous one. □

Similar invariants hold for the delivery of safe messages.

Invariant 4.21 (DVS-IMPL)
In any reachable state s, we have that if $s.safe\text{-}from\text{-}vs[g]_p = \langle\langle m_1,q_1\rangle, \langle m_2,q_2\rangle, ..., \langle m_k,q_k\rangle\rangle$, then $\mathcal{F}(s).queue[g](next\text{-}safe[p,g], next\text{-}safe[p,g] + k - 1) = \langle\langle m_1,q_1\rangle, \langle m_2,q_2\rangle, ..., \langle m_k,q_k\rangle\rangle$.

Proof: The proof is as for msgs except that it uses the *safe-from-vs* queue instead of *msgs-from-vs* and the pointer *next-safe* instead of *next.* □

Invariant 4.22 (DVS-IMPL)
In any reachable state s, if $\langle m,q\rangle$ is head of $s.safe\text{-}from\text{-}vs[g]_p$, then $\mathcal{F}(s).queue[g](next\text{-}safe[p,g]) = \langle m,q\rangle$.

Proof: Follows easily from previous one. □

Notice that v is totally registered in state s of DVS-IMPL if and only if it is totally registered in the state of DVS that appears in state $\mathcal{F}(s)$ of DVS.

4.3.2.2 Proof that $\mathcal{F}$ is an abstraction function

In order to prove that $\mathcal{F}$ is an abstraction function we need to prove that (a) for any initial state s of DVS-IMPL we have that $\mathcal{F}(s)$ is an initial state of DVS, and that (b) for any possible step π of DVS-IMPL there exists an execution fragment α of DVS such that the trace of α is equal to the trace of π, that is, α and π have identical externally observable behaviors. Lemmas 4.23 and 4.24 prove this.

Lemma 4.23 *If s is an initial state of* DVS-IMPL *then $\mathcal{F}(s)$ is an initial state of* DVS.

Proof: Let s_0 be the unique initial state of DVS-IMPL and t_0 the unique initial state of DVS.

We have $s_0.attempted_p = \{v_0\}$ for $p \in P_0$ and $s_0.attempted_p = \{\}$ for $p \notin P_0$. By the definition of $\mathcal{F}$ and the fact that $P_0 \neq \{\}$ (because all membership sets are defined to be nonempty), we have $\mathcal{F}(s_0).created = \{v_0\}$. This is as in t_0.

We have $s_0.client\text{-}cur_p = \{v_0\}$ for $p \in P_0$ and $s_0.client\text{-}cur_p = \perp$ for $p \notin P_0$. By the definition of $\mathcal{F}$ we have $\mathcal{F}(s_0).current\text{-}viewid[p] = g_0$ for $p \in P_0$ and $\mathcal{F}(s_0).current\text{-}viewid[p] = \perp$ for $p \notin P_0$. This is as in t_0.

We have $s_0.attempted_p = \{v_0\}$ for $p \in P_0$ and $s_0.attempted_p = \{\}$ for $p \notin P_0$. By the definition of $\mathcal{F}$ we have $\mathcal{F}(s_0).attempted[g_0] = P_0$ and $\mathcal{F}(s_0).attempted[g] = \{\}$ for $g \neq g_0$. This is as in t_0.

Let $g \in \mathcal{G}$. We have that $s_0.reg[g]_p$ is `true` if and only if $p \in P_0$ and $g = g_0$. By the definition of $\mathcal{F}$ we have $\mathcal{F}(s_0).registered[g_0] = P_0$ and $\mathcal{F}(s_0).registered[g] = \{\}$ for $g \neq g_0$, as in t_0.

Let $p \in \mathcal{P}$. We have that $s_0.msgs\text{-}to\text{-}vs[g]_p = \lambda$ and $s_0.pending[p, g] = \lambda$. By the definition of $\mathcal{F}$ we have $\mathcal{F}(s_0).pending[p, g] = \lambda$, as in t_0.

Let $g \in \mathcal{G}$. We have $s_0.queue[g] = \lambda$. By the definition of $\mathcal{F}$ we have $\mathcal{F}(s_0).queue[g] = \lambda$, as in t_0.

Let $p \in \mathcal{P}, g \in \mathcal{G}$. We have $s_0.next[p, g] = 1$, $purgesize(s.\text{VS}.queue[g]) = 0$ and $s_0.msgs\text{-}from\text{-}vs[g]_p = \lambda$. By the definition of $\mathcal{F}$ we have $\mathcal{F}(s_0).next[p, g] = 1$, as in t_0. A similar argument holds for *next-safe*.

Thus $\mathcal{F}(s_0) = t_0$, as needed. □

Lemma 4.24 *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$(v)_p$.) Define $t = \mathcal{F}(s)$ and $t' = \mathcal{F}(s')$.

1. $\pi =$ VS-CREATEVIEW(v)

   Then $trace((s, \pi, s')) = \lambda$. Action π modifies *created*. The definition of $\mathcal{F}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

2. $\pi =$ VS-NEWVIEW$(v)_p$

   Then $trace((s, \pi, s')) = \lambda$. Action π modifies cur_p, $info\text{-}sent[cur.id]_p$, and $current\text{-}viewid[p]$, and adds an *"info"* message to $msgs\text{-}to\text{-}vs[cur.id]_p$. The definition of $\mathcal{F}$ is not sensitive to any of these changes. Therefore, $t = t'$, and we set $\alpha = t$.

3. $\pi = \text{VS-GPSND}(m)_p$

   Then $\mathit{trace}((s, \pi, s')) = \lambda$. Action π just moves a message from the queue $\mathit{msgs\text{-}to\text{-}vs}[\mathit{cur.id}]_p$ to the queue $\mathit{pending}[p, \mathit{current\text{-}viewid}[p]]$. The definition of $\mathcal{F}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

4. $\pi = \text{VS-ORDER}(m,p,g)$

   Then $\mathit{trace}((s, \pi, s')) = \lambda$. Action π moves a message from $\mathit{pending}[p, g]$ to $\mathit{queue}[g]$. We consider two cases.

   (a) $m \in \mathcal{M}_c$

   Then we set $\alpha = (t, \text{DVS-ORDER}(m,p,g), t')$. We claim that $\text{DVS-ORDER}(m,p,g)$ is enabled in t: Since $\text{VS-ORDER}(m,p,g)$ is enabled in s, it follows that m is the head of $s.\mathit{pending}[p, g]$. By the definition of $\mathcal{F}$, m is also the head of $t.\mathit{pending}[p, g]$. It follows that $\text{DVS-ORDER}(m,p,g)$ is enabled in t.

   By definition of $\mathcal{F}$, t' differs from t only in the fact that m is moved from $\mathit{pending}[p, g]$ to $\mathit{queue}[g]$. This is the effect achieved by applying $\text{DVS-ORDER}(m,p,g)$ to t.

   (b) $m \notin \mathcal{M}_c$

   Then the definition of $\mathcal{F}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

5. $\pi = \text{VS-GPRCV}(\langle \text{"}\mathit{info}\text{"}, v, s\rangle)_{q,p}$

   Then $\mathit{trace}((s, \pi, s')) = \lambda$. This action can modify $\mathit{info\text{-}rcvd}[\mathit{cur.id}_p, q]_p$, act_p and amb_p (see code of DVS) and causes $\mathit{next}[p, \mathit{cur.id}_p]$ to be incremented (see code of VS). The definition of $\mathcal{F}$ is not sensitive to these changes. (The only interesting case is the definition of $t.\mathit{next}[p, \mathit{cur.id}_p]$, where the absolute values of the first two terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

6. $\pi = \text{VS-GPRCV}(\text{"}\mathit{registered}\text{"})_p$

   Then $\mathit{trace}((s, \pi, s') = \lambda$. This action can modify $\mathit{rcvd\text{-}rgst}[\mathit{cur.id}, q]_p$. It also causes the pointer $\mathit{next}[p, \mathit{cur.id}_p]$ to be incremented. The definition of $\mathcal{F}$ is not sensitive to these changes. (The only interesting case is the definition of $t.\mathit{next}[p, \mathit{cur.id}_p]$, where the absolute values of the first two terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

7. $\pi = \text{VS-GPRCV}(m)_p$, $m \in \mathcal{M}_c$

   Then $\mathit{trace}((s, \pi, s')) = \lambda$. This action copies a message from the sequence $\mathit{queue}[\mathit{cur.id}]_p$ to the sequence $\mathit{msgs\text{-}from\text{-}vs}[p, \mathit{client\text{-}cur}[p]]$, and causes $\mathit{next}[p, \mathit{cur.id}_p]$ to be incremented. The definition of $\mathcal{F}$ is not sensitive to these changes. (The only interesting case is the definition of $t.\mathit{next}[p, \mathit{cur.id}_p]$, where the absolute values of the first and third terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

8. $\pi = \text{VS-SAFE}(\langle m, v, s\rangle)_{q,p}$, $m \in \{\text{"}\mathit{info}\text{"}, \text{"}\mathit{registered}\text{"}\}$

Then $trace((s, \pi, s')) = \lambda$. Action π just causes $next\text{-}safe[p, cur.id_p]$ to be incremented. The definition of $\mathcal{F}$ is not sensitive to this change. (The only interesting case is the definition of $t.next\text{-}safe[p, cur.id_p]$, where the absolute values of the first two terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

9. $\pi =$ VS-SAFE$(m)_p$, $m \in \mathcal{M}_c$

   Then $trace((s, \pi, s')) = \lambda$. Action π adds a message to $safe\text{-}from\text{-}vs[cur.id]_p$ and causes the pointer $next\text{-}safe[p, cur.id_p]$ to be incremented. The definition of $\mathcal{F}$ is not sensitive to these changes. (The only interesting case is the definition of $t.next\text{-}safe[p, cur.id_p]$, where the absolute values of the first and third terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

10. $\pi =$ DVS-NEWVIEW$(v)_p$

    Then $trace((s, \pi, s)) = \pi$. In DVS-IMPL, this action modifies only variables amb_p, $attempted_p$, $client\text{-}cur_p$. We have $s'.client\text{-}cur_p = v$ and $s'.attempted_p = s.attempted_p \cup \{v\}$. By definition of $\mathcal{F}$, we have that $t'.current\text{-}viewid[p] = s'.client\text{-}cur.id_p = v.id$, $t'.created = t.created \cup \{v\}$ and $t'.attempted[v.id] = t.attempted[v.id] \cup \{p\}$, while all other state variables in t' are as in t.

    We consider two cases:

    (a) $v \in t.created$.

    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 $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$.

    (b) $v \notin t.created$.

    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 $\exists x \in s.TotReg$ 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 4.1 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 $x \in s.TotReg$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$. Since $w \in t.created$, by definition of $\mathcal{F}$, $w \in s.attempted_q$ for some q. Clearly, $w \in s'.attempted_q$. Therefore, $w \in s'.Att$. By the code of π we have that $v \in s'.attempted_p$. Therefore we also have $v \in s'.Att$. Moreover, there is no $x \in s'.\mathit{TotReg}$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$. Then Invariant 4.18 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''.

11. $\pi =$DVS-REGISTER$_p$

    Then $trace((s, \pi, s')) = \pi$. Let g be $s.client\text{-}cur.id_p$, which equals $t.current\text{-}viewid[p]$ by the abstraction function. If $g = \bot$, then π has no effect in DVS-IMPL, so $s = s'$; thus $t = t'$, as required to show that π brings DVS from t to t'. Otherwise, $g \neq \bot$, so by the code in DVS-IMPL, this action sets $reg[g]_p$ to `true` and inserts a *"registered"* message into $msgs\text{-}to\text{-}vs[g]_p$. By definition of $\mathcal{F}$, t' is the same as t except that $t'.registered[g] = t.registered[g] \cup \{p\}$. We set $\alpha = (t, \text{DVS-REGISTER}_p, t')$. It is easy to check that DVS-REGISTER$_p$ brings DVS from t to t'.

12. $\pi =$DVS-GARBAGECOLLECT$(v)_p$

    Then $trace((s, \pi, s')) = \lambda$. This action can modify act_p and amb_p. The definition of $\mathcal{F}$ is not sensitive to these changes. Therefore, $t = t'$, and we set $\alpha = t$.

13. $\pi =$DVS-GPSND$(m)_p$

    Then $trace((s, \pi, s')) = \pi$. We set $\alpha = (t, \text{DVS-GPSND}(m)_p, t')$. We consider two cases:

    (a) $s.client\text{-}cur.id = \bot$

    Then $s = s'$. In this case, the definition of $\mathcal{F}$ implies that also $t.current\text{-}viewid[p] = \bot$, which implies that the action also has no effect in t, which suffices.

    (b) $s.client\text{-}cur.id \neq \bot$

    In this case, the action appends m to $msgs\text{-}to\text{-}vs[g]_p$, where $g = client\text{-}cur.id_p$. Hence we have that $s'.msgs\text{-}to\text{-}vs[g] = s.msgs\text{-}to\text{-}vs[g]+m$. By the definition of $\mathcal{F}$ we get that $t'.pending[p, g] = t.pending[p, g]+m$. This is the effect of the action in t (using the fact that $t.current\text{-}viewid[p] \neq \bot$.)

14. $\pi =$ DVS-GPRCV$(m)_p$

    Then $trace((s, \pi, s')) = \pi$. This action removes the head of $msgs\text{-}from\text{-}vs[g]_p$, where $g = cur.id_p$. We have that $s.msgs\text{-}from\text{-}vs[g]_p = m+s'.msgs\text{-}from\text{-}vs[g]_p$. Thus $t'.next[p, g] = t.next[p, g] + 1$. We set $\alpha = (t, \text{DVS-GPRCV}(m)_p, t')$. It is easy to check that the step has the required effect in DVS. The fact that DVS-GPRCV$(m)_p$ is enabled in t follows from Invariant 4.20.

15. $\pi =$ DVS-SAFE$(m)_p$

    Then $trace(\pi) = \pi$. This action removes the head of the $safe\text{-}from\text{-}vs[g]_p$, where $g = cur.id_p$. We have that $s.safe\text{-}from\text{-}vs[g]_p = m+s'.safe\text{-}from\text{-}vs[g]_p$. Thus $t'.next\text{-}safe[p, g] = t.next\text{-}safe[p, g]+$

1. We set $\alpha = (t, \text{DVS-GPRCV}(m)_p, t')$. It is easy to check that the step has the required effect in DVS. The fact that $\text{DVS-GPRCV}(m)_p$ is enabled in t follows from Invariant 4.22.

□

Lemmas 4.23 and 4.24 prove that $\mathcal{F}$ is an abstraction function from DVS-IMPL to DVS and thus the following theorem holds (this is a standard inference, cf. [29]).

Theorem 4.25 *Every trace of* DVS-IMPL *is a trace of* DVS.

5 An application of DVS

Now we demonstrate the utility of DVS by showing how to use it to implement a totally ordered broadcast service, called TO, originally defined in [17]. This service accepts messages from clients and delivers them to all clients according to the same total order. This kind of service is used as a building block for many fault-tolerant distributed applications, e.g., in implementing sequentially-consistent shared memory and atomic shared memory. The TO specification is reproduced in Figure 6.

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

Transitions:

```
input BCAST(a)_p
  Eff: append a to pending[p]

internal TO-ORDER(a,p)
  Pre: a is head of pending[p]
  Eff: remove head of pending[p]
       append ⟨a,p⟩ to queue

output BRCV(a)_{p,q}
  Pre: queue(next[q]) = ⟨a,p⟩
  Eff: next[q] := next[q] + 1
```

Figure 6: The TO service

5.1 The implementation TO-IMPL

We provide an implementation of TO using DVS as a building block. The implementation is similar to the TO implementation provided in [17]. Both algorithms rely on primary views to establish a total order of client messages. The difference is that the algorithm in [17] uses a static notion of primary and our new algorithm uses a dynamic notion. The algorithm of [17] is built upon the VS service, defined in the same paper, that reports non-primary as well as primary views; that algorithm uses a simple local test to determine if the view is primary, namely it checks whether the view contains a majority of the processes; the algorithm also does some non-critical background work (gossiping information) in non-primary views. In contrast, the algorithm we present here is built upon the 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 that they have obtained whatever information they need to proceed with regular computation in the new view. The corresponding notion in [17] is that of an *established* view: an established view in the algorithm of [17] corresponds to a *registered* view in our algorithm. Although the new algorithm appears very similar to the one of [17], the fact that the DVS service provides more complicated guarantees than the VS service makes the new algorithm harder to prove correct.

The TO-IMPL 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 consists of a view identifier (the one of the view in which the message is received), a sequence number and the identifier of the process receiving the message. The association between client messages and their unique labels is recorded 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. When client messages have been reported as delivered to all the members of the view, by the "safe" notification of DVS, the label and its order may become *confirmed*. The messages associated with confirmed labels may be released to the clients in the given order.

The consistent sequence of message delivery within each view keeps this tentative order consistent at members of a given view, but it may be not consistent between processes in different views. To avoid inconsistencies processes need state exchange at the beginning of a new view.

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. The summary of a node's state contains the following information: the association of labels with client messages, stored in *content*, the order of client messages to be reported to the clients, stored in *order*, a pointer to the next client message to be confirmed, stored in *nextconfirm* and the view identifier of the primary view with the highest view identifier in which the *order* sequence has been modified (stored in *highprimary*).

Once a node has received all members' state summaries, it processes the information in one atomic step, i.e., it registers the new view using the DVS-REGISTER action. The node processes state information as follows: it defines its confirmed labels to be the longest prefix of confirmed labels known in any of the summaries; it determines the *representatives* as the members whose summary include the greatest *highprimary* value; adopts as its new *order* the *order* of a "chosen" representative (the chosen representative is arbitrary but must be the same for all processes) extended with all other labels appearing in any of the received summaries, arranged in label order.

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

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*, *established*$\}$,
init *normal*
content $\in 2^{\mathcal{C}}$, init $\{\}$
nextseqno $\in \mathbf{N}^{>0}$, init 1
buffer $\in$ *seqof*$(\mathcal{L})$, init λ
safe-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 if $p \in P_0$, $\perp$ else
gotstate, a partial function from $\mathcal{P}$ to $\mathcal{S}$, init $\{\}$
safe-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

Figure 7: $\textsc{dvs-to-to}_p$, signature and states

just as in normal processing.

For the code, shown in Figures 7 and 8, we need the following definitions. $\mathcal{L} = \mathcal{G} \times \mathbf{N}^{>0} \times \mathcal{P}$ 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*. Given $x \in \mathcal{S}$, *x.confirm* is the prefix of *x.ord* such that $|x.confirm| = min(x.next - 1, |x.ord|)$. If Y is a partial function from processor ids to summaries, then we define:

- $knowncontent(Y) = \cup_{q \in dom(Y)} Y(q).con$,
- $maxprimary(Y) = \max_{q \in dom(Y)}\{Y(q).high\}$,
- $maxnextconfirm(Y) = \max_{q \in dom(Y)} Y(q).next$,
- $reps(Y) = \{q \in dom(Y) : Y(q).high = maxprimary\}$,
- $chosenrep(Y) =$ 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.

We define the system TO-IMPL to be the composition of the automata $\textsc{dvs-to-to}_p$, for each $p \in \mathcal{P}$, and the DVS specification. with all the external actions of DVS hidden.

Following the approach in [17], we define the derived variables *allstate*, *allcontent* and *allconfirm* for TO-IMPL as follows.

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

Transitions:

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

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

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
         $status := established$

output DVS-REGISTER$_p$
  Pre: $status = established$
       $current.id \notin registered$
  Eff: $registered := registered \cup \{current.id\}$
         $nextconfirm := maxnextconfirm(gotstate)$
         $order := fullorder(gotstate)$
         $highprimary := current.id$
         $status := normal$

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: DVS-TO-TO$_p$, transitions

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

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

- We write *allcontent* for $\bigcup_{x \in allstate} x.con$

  This represents all the information available anywhere that links a label with a corresponding data value.

- We write *allconfirm* for $lub_{x \in allstate}(x.confirm)$.

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

5.2 Correctness proof

The correctness proof for TO-IMPL follows the approach of the proof in [17]. The pattern of reasoning is the same as the one used in that proof, however there are differences due to the distinct guarantees offered by DVS compared to those of VS. In particular Invariant 5.6, which corresponds to Lemma 6.18 of [17], requires a more subtle proof.

We start by providing some auxiliary invariants.

Invariant 5.1 (TO-IMPL)
In any reachable state, if $l \in domain(allcontent)$ and $l.origin = p$ then $l < \langle current.id_p, nextseqno_p, p\rangle$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, no label is associated with any message hence the invariant is vacuously true.

For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, s'). The only step that can make the assertion false is the step when a new label is associated with a message from a client, hence we only need to consider $\pi = \text{LABEL}(a)_p$. The code of π shows that the new label is less than the new $\langle current.id_p, nextseqno_p, p\rangle$ triple, since $nextseqno_p$ is incremented after being used to create the label. □

The following invariant says that when a process p has registered a view v, then any summary that p will create for a later view w will have its *highprimary* component equal to v or to a later view.

Invariant 5.2 (TO-IMPL)
In any reachable state, let x be a summary, $p \in P$, and $v, w \in created$ such that $v.id \in registered_p$, $w.id > v.id$, and $x \in allstate[p, w.id]$. Then then $x.high \geq v.id$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, there are no v and w that satisfy the hypothesis. Hence the invariant is vacuously true.

For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, s'). The steps that can make the assertion false are those that create new summaries or new views.

Let us first consider actions that create new summaries. When a new summary is created without modifying the *high* component of existing summaries in $allstate[p, w.id]$, then the assertion cannot be made false. So we only need to consider action $\pi =$DVS-REGISTER$_p$ when $current_p = w.id$, which creates the first summary x that satisfy the hypothesis in s' (no such x existed in s). In this case we have that $x'.high = w.id$ and by the inductive hypothesis $w.id > v.id$, hence we have that $x'.high > v.id$.

Consider now actions that create new views, that is $\pi =$DVS-NEWVIEW$(w)_p$. We distinguish two cases: (1) the only registered view is v_0, (2) there are registered views other than v_0. In case (1) we have that $y.high = g_0$ for any existing summary y; hence the invariant is true. In case (2) the invariant is true by inductive hypothesis. □

Next we provide some other auxiliary invariants.

Invariant 5.3 (TO-IMPL)
In any reachable state, if $x \in allstate[p, g]$ then there exists $v \in created$ and $q \in v.set$ such that: (1) $x.high = v.id$, (2) $x.high \in registered_q$, (3) $x.ord = buildorder[q, x.high]$ and (4) either $x.high = g$ or $current.id_q > v.id$

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, for any such x we have that $x.high = g_0$ and $x.ord = \lambda$ and thus we can take $v = v_0$ and $q = p$ and the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, s'). If $x \in s'.allstate[p, g]$, then in most cases, there is $y \in s.allstate[p, g]$ with $y.high = x.high$ and $y.ord = x.ord$, to which we apply the induction hypothesis. The only case where this does not happen is when $\pi =$DVS-NEWVIEW$(v)_p$, where $v.id = g$, and x is the summary whose components are taken from the state of p. In this case, there is $y \in s.allstate[p, s.current_p]$ with $y.high = x.high$ and $y.ord = x.ord$, to which we apply the inductive hypothesis. □

Invariant 5.4 (TO-IMPL)
In any reachable state, if $x \in allstate$ then there exists $w \in created$ such that $x.high = w.id$, and for all $p \in w.set$, $p \in attempted[w.id]$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, the invariant follows from the definition of *allstate* (set $w = current.id_p$).
For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, s'). The only step that we have to worry about is when a new summary is created. When a new summary x is created, $x.high$ is set to the identifier of the current view, and a message has been received from everyone in the membership. □

Invariant 5.5 (TO-IMPL)
In any reachable state, 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. Invariant 5.4 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 3.4 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. □

Next we provide the crucial invariant corresponding to Lemma 6.18 of [17]. This invariant has a more subtle proof than the one given in Lemma 6.18 of [17]. That proof 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 totally attempted, no views with lower identifiers can be registered.

Invariant 5.6 (TO-IMPL)
In any reachable state, 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 $v.id \in registered_p$ and $\sigma \leq buildorder[p, v.id]$. Then for every $x \in allstate$ with $x.high > v.id$, we have that $\sigma \leq x.ord$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. 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.

For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, 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 $v.id \in s'.registered_p$ and $\sigma \leq s'.buildorder[p, v.id]$.

If $v \notin s.created$, then π must be CREATEVIEW(v). Then $v.id \notin s'.registered_p$ for all p. Fix $x \in s'.allstate$ and suppose that $x.high > v.id$. Then Invariant 5.5 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 $v.id \in s'.registered_p$, a contradiction. It follows that $v \in s.created$.

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

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

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

Any step that produces the new summary x by modifying an old summary $x' \in s.allstate$, in such a way that $x'.ord \leq x.ord$ and $x'.high = x.high$, is easy to handle: For such a step, $x'.high > v.id$ and so the inductive hypothesis implies that $\sigma \leq x'.ord \leq x.ord$, as needed. So the only concern is with DVS-REGISTER$_p$ steps that produce a new summary x from the state-exchange messages of a view w sent 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 view identifier among the elements of $s'.created$ such that $v'.id < w.id$ and v' is totally registered in s'. Let v'' denote either v' or v, whichever has the higher view identifier. Invariant 3.3 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 precondition $status = established$ of DVS-NEWVIEW implies that $domain(s'.gotstate_p) = w.set$, so by the code $q'' \in domain(s.gotstate_p)$. Let x'' be the summary $s.gotstate(q'')_p$; we have $x'' \in s.allstate[q'', w.id]$.

We now show that $v''.id \in s.registered_{q''}$. We consider two cases:

1. $v'' = v'$.

   Then $q'' \in v'.set$ so by definition of v', we have that $v'.id \in s.registered_{q''}$.

2. $v'' = v$. Because $s.allstate[q'', w.id]$ is non-empty, we have 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 invariant is true. Therefore, by this hypothesis (uses $q'' \in v.set$), we obtain that $v.id \in s.registered_{q''}$

By Invariant 5.2 (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.

If $x'.high > v.id$ then we can apply the inductive hypothesis to x' and we are done, since $x'.ord \leq x.ord$. So suppose $x'.high = v.id$. Note that $x' \in s.allstate[q', w.id]$. By Invariant 5.3 there must exist[2] $q \in v.set$ so that $v.id \in s.registered_q$, $x'.ord = s.buildorder[q, v.id]$, and (either $x'.high = w.id$ or $s.current.id_q > v.id$). Since $x'.high = v.id < x.high = w.id$, the last property can be simplified to $s.current.id_q > v.id$. By monotonicity of $current$, we have $s'.current_q > v.id$. The hypothesis of this invariant says that this forces $\sigma \leq s'.buildorder[q, v.id]$. Since $x'.ord \leq x.ord$ by the code for this event, and $x'.ord = s.buildorder[q, v.id]$ as shown above, and $s.buildorder[q, v.id] = s'.buildorder[q, v.id]$ since q is not currently in view v, we get $\sigma \leq x.ord$, which is what we need. □

Next we provide some additional auxiliary invariants.

Invariant 5.7 (TO-IMPL)
In any reachable state, if we have that $v \in created$, $\sigma \in seqof(\mathcal{L})$, and for every $p \in v.set$, $v.id \in registered_p$ and $\sigma \leq buildorder[p, v.id]$, then for every $x \in allstate$ with $x.high \geq v.id$, $\sigma \leq x.ord$.

Proof: The are two possible cases: (1) $x.high > v.id$, (2) $x.high = v.id$. In case (1) we can apply Lemma 5.6. Consider case (2). Then we apply Lemma 5.3 to x, which gives $v' \in created$ and $q' \in v'.set$ such that $x.high = v'.id$, $x.high \in registered_{q'}$, and $x.ord = buildorder[q', x.high]$. Since $v.id = v'.id$, Lemma 4.1 shows $v = v'$. Substituting in the facts above we see $x.ord = buildorder[q', v.id]$. Since $q' \in v.set$, we can apply the premise of the corollary to see that $\sigma \leq buildorder[q', v.id]$; that is, $\sigma \leq x.ord$, as required. □

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

Invariant 5.8 (TO-IMPL)
In any reachable state, if $l \in safe\text{-}labels_p$ and σ is a prefix of $order_p$ that is terminated by l, then for all $q \in current.set_p$, $\sigma \leq buildorder[q, current.id_p]$

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

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

Invariant 5.9 (TO-IMPL)
In any reachable state, the following is true. Assume $l, l' \in \mathcal{L}$ and $i' \in \mathbf{N}^{>0}$. If $l, l' \in domain(allcontent)$ and $l.origin = l'.origin$ and $l < l'$ and $x \in allstate$ and $l' = x.ord(i')$ then there exists i such that $i < i' \wedge l = x.ord(i)$

The proofs of Lemmas 5.8 and 5.9 are left as exercises. Next we show that *x.confirm* is a prefix of a known sequence. This shows the consistency of the confirmed sequence of labels at different places in the system.

Invariant 5.10 (TO-IMPL)
In any reachable state, if $x \in allstate$ then

1. *There exists $v \in created$ such that $v.id \le x.high$ and for every $q \in v.set$, $v.id \in registered_q$ and $x.confirm \le buildorder[q, v]$.*

2. *$x.next \le length(x.ord) + 1$*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, the only created view is v_0 and the only extant summary is $\langle \{\}, \lambda, 1, g_0 \rangle$; it is easy to verify that the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, s').

For most of the steps, there is y in *s.allstate* so that $y.next = x.next$, $y.ord = x.ord$ (and hence $y.confirm = x.confirm$) and also $y.high = x.high$. In these cases, the inductive hypothesis gives us what we want, since $buildorder[q, v]$ increases monotonically through an execution.

The steps that are left to consider are CONFIRM$_p$, DVS-GPRCV$(\langle l, a \rangle)_{q,p}$ and DVS-REGISTER$_p$.

Consider the case π=CONFIRM$_p$. If x is not the summary from the state of p in s', the invariant follows from the inductive hypothesis. If x is the summary from the state of p in s', the precondition of π shows that the newly confirmed message has label in $s.safe\text{-}label_p$. By Invariant 5.8, taking v to be $s.current_p = x.high$, we have part 1 of the invariant. The precondition of π also gives $(x.next - 1) \in domain(x.ord)$, thus showing part 2 of the invariant.

Now consider π =DVS-GPRCV$(\langle l, a \rangle)_{q,p}$. As before, if x is not the summary from the state of p in s', the invariant follows from the inductive hypothesis. If x is the summary from the state of p, let y denote the summary taken from p in state s. The code shows that $x.high = y.high$, $x.next = y.next$, and $x.ord$ is an extension of $y.ord$. By part 2 applied to y, we see that $y.next \le length(y.ord) + 1$ and therefore $x.next \le length(x.ord) + 1$. Which proves part 2 for x; also it shows that $x.confirm = y.confirm$, so that the inductive hypothesis of part 1 applied to y proves part 1 for x.

Finally consider π =DVS-REGISTER$_p$. As in the other two cases, if x is not the summary from the state of p in s', the invariant follows from the inductive hypothesis. If x is the summary from the state of p, let y denote the summary, among those in $gotstate_p$, with the highest value for $y.next$. The code shows that $x.next = w.next$. Summary y is in *s.allstate*. The inductive hypothesis shows that *y.confirm* has length $y.next - 1$, and that there is $v \in s.created$ such that $v.id \le y.high$

and $\forall q \in v.set$ it holds $v.id \in s.established_q$ and $y.confirm \leq buildorder[q, v]$). Now let z denote the summary of $chosenrep(gotstate)$, as calculated in the effect of π. Since $z.high \geq y.high \geq v.id$ (recall the definition of z as being from a representative, that is, having maximal highprimary among summaries in $gotstate$), Invariant 5.7 shows that $y.confirm \leq z.ord$. Since $z.ord \leq x.ord$ by the code, we deduce that $y.confirm$ is a prefix of $x.ord$; as $length(y.confirm) = y.next - 1 = x.next - 1$, we have $x.confirm = y.confirm$. Also by the code we have $x.high \geq y.high$. Thus the inductive hypothesis applied to y, along with the monotonicity of the set *created* and the fact that $v.id \in registered_q$, gives the invariant for x. □

Invariant 5.11 (TO-IMPL)
In any reachable state, if $x_1, x_2 \in allstate$ and $x_1.high \leq x_2.high$, then $x_1.confirm \leq x_2.ord$.

Proof: By Invariant 5.10, part 1, with $x = x_1$, we have that there exists v such that $v.id \leq x_1.high$ and $x_1.confirm \leq buildorder[q, v]$. By Invariant 5.7 used with $\sigma = x_1.confirm$ since $x_2.high \geq v.id$ we have that the conclusion of Invariant 5.7 holds for x_2. Hence $x_1.confirm \leq x_2.ord$. □

Invariant 5.12 (TO-IMPL)
In any reachable state, for any $x, x' \in allstate$, either $x.confirm \leq x'.confirm$ or $x'.confirm \leq x.confirm$.

Proof: Without loss of generality, we can assume that $x.high \leq x'.high$. From Invariant 5.11, we have that both $x.confirm$ and $x'.confirm$ are prefixes of $x'.order$. □

To prove that TO-IMPL implements TO, we define a function from the reachable states of TO-IMPL to the states of TO and prove that it is an abstraction function. This function, called $\mathcal{F}_{TO}$, is defined exactly as in [17] and it is given in Figure 9.

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

1. $t.queue = applyall(\langle s.allcontent, origin \rangle, s.allconfirm)$,
   where the selector *origin* is regarded as a function from labels to processors.

2. $t.next[p] = s.next\text{-}report_p$.

3. $t.pending[p] = applyall(s.allcontent, b) \cdot s.delay_p$ where b is the sequence of labels such that

   (a) $range(b)$ is the set of labels l such that $l.origin = p$, $\langle l, a \rangle \in s.allcontent$ for some a, and
   $l \notin range(s.allconfirm)$.

   (b) b is ordered according to the label order.

Figure 9: The abstraction function $\mathcal{F}_{TO}$.

In order to prove that $\mathcal{F}_{TO}$ is an abstraction function we need to prove that (a) for any initial state s of TO-IMPL we have that $\mathcal{F}_{TO}(s)$ is an initial state of TO, and that (b) for any possible step π of TO-IMPL there exists an execution fragment α of TO such that the trace of α is equal to the trace of π, that is, α and π have identical externally observable behaviors. Lemmas 5.13 and 5.14 prove this.

Lemma 5.13 *If s is an initial state of* TO-IMPL *then $\mathcal{F}(s)$ is an initial state of* TO.

Proof: Let s be the initial state of TO-IMPL. Let $t = \mathcal{F}_{TO}(s)$. By the definition of $\mathcal{F}_{TO}$ we have that $t.queue = \lambda$, $t.next[p] = s.next\text{-}report_p = 1$ for any p and that $t.pending = \lambda$. Hence t is an initial state of TO. □

Lemma 5.14 *Let s be a reachable state of* TO-IMPL, *$\mathcal{F}_{TO}(s)$ a reachable state of* TO, *and (s, π, s') a step of* TO-IMPL. *Then there is an execution fragment α of* TO *that goes from $\mathcal{F}_{TO}(s)$ to $\mathcal{F}_{TO}(s')$, such that $trace(\alpha) = trace(\pi)$.*

Proof: By case analysis based on the type of the action π. Define $t = \mathcal{F}_{TO}(s)$ and $t' = \mathcal{F}_{TO}(s')$.

1. $\pi = \text{BCAST}(a)_p$

   Since π is an input to TO, π is enabled in t. The effect of π shows that $s'.allconfirm = s.allconfirm$, $s'.allcontent = s.allcontent$, and $s'.pending[p] = s.pending[p] + a$. This implies that $t'.pending[p] = t.pending[p] + a$, thus showing that (t, π, t') is a step of TO. Hence we set $\alpha = \pi$. Clearly $trace(\alpha) = trace(\pi)$.

2. $\pi = \text{LABEL}(a)_p$

   We to show that $t = t'$. The effect of π shows that $t'.allconfirm = x.allconfirm$, and $t'.allcontent$ is the union of $t.allcontent$ with $\langle l', a\rangle$ where $l' = \langle t.current_p, x.nextseqno_p, p\rangle$; by Invariant 5.1, this new label l' is greater than all labels in the domain of $x.allcontent$. Thus let us consider the sequence of labels σ' (arranged in label order) such that $range(\sigma')$ is the set of labels l such that $l.origin = p$, $\langle l, a'\rangle \in x'.allcontent$ for some a', and $l \notin range(x'.allconfirm)$. We see that σ' is related to the sequence σ (defined the same way but using s instead of s') by $\sigma' = \sigma + l'$. Therefore $applytoall(s'.allcontent, \sigma') = applytoall(s.allcontent, \sigma) + a$. On the other hand, the precondition of π shows that a is the head of $s.delay_p$, and so the effect of π means $s.delay_p = a + x'.delay_p$. Thus, $t'.pending[p]$ is the same as $t.pending[p]$, because in the concatenation that defines this component, the element a is simply transferred from suffix to prefix. Therefore $t' = t$. Hence we set $\alpha = t$.

3. $\pi = \text{CONFIRM}_p$

   Clearly the effect of π shows $s.allcontent = s'.allcontent$.

   If $s.nextconfirm_p \leq length(s.allconfirm)$ then Invariant 5.12 and the effect of π shows that $s'.allconfirm = s.allconfirm$, so that $t = t'$. In this case we set $\alpha = t$.

   Otherwise $s.nextconfirm_p = length(s.allconfirm) + 1$, so the effect of π shows that $s'.allconfirm = x.allconfirm \cdot \langle l\rangle$ where $l = s.order_p(s.nextorder_p)$. Let $q = l.origin$ and $a = s.allcontent(l)$. We claim that $(t, \text{TO-ORDER}(a, q), t')$ is a step of TO.

   We first show that $\text{TO-ORDER}(a, q)$ is enabled in t. We have $l \in domain(s.allcontent)$ and $l \notin setof(s.allconfirm)$; this means that a is an element of the sequence $t.pending[q]$. Also by Invariant 5.9, any lower label with origin q is in $s.confirm_p$ and so in $s.allconfirm$. Since the sequence σ used to define $t.pending[q]$ is arranged by label, we see that l is the head of σ, and so a is the head of $t.pending[q]$, as required. Further, the equation above for $t'.allconfirm$ shows that $t'.queue = t.queue + \langle a, p\rangle$, and this is what is needed to show that π takes t to t'.

   Hence we set $\alpha = \text{TO-ORDER}(a, q)$. Clearly $trace(\alpha) = trace(\pi) = \lambda$.

4. $\pi =$ DVS-GPRCV$(s)_{q,p}$

   In some cases this may change the value of $nextconfirm_q$, but in every situation it leaves *allconfirm* unchanged (it only moves $nextconfirm_q$ to a value already somewhere in *allstate*). Thus $t' = t$. Hence we set $\alpha = t$.

5. $\pi =$ BRCV$(a)_{p,q}$

   We need to show that π is enabled in t as an action of TO. This is immediate from the fact that π is enabled in s as an action of TO-IMPL. Similarly, the effect corresponds (only $nextreport_q$ is altered).

   Hence we set $\alpha =$BRCV$(a)_{q,p}$. Clearly $trace(\alpha) = trace(\pi) = \lambda$.

6. Remaining actions.

   The other actions leave $t' = t$. Hence we set $\alpha = t$.

□

Lemmas 5.13 and 5.14 prove that $\mathcal{F}_{TO}$ is an abstraction function from TO-IMPL to TO and thus the following theorem holds (this is a standard inference, cf. [29]).

Theorem 5.15 *Every trace of* TO-IMPL *is a trace of* TO.

6 Discussion

We presented a specification for a dynamic primary view group communication service and an algorithm that formally implements the service, and we showed the utility of our new specification by using it to implement a totally ordered broadcast. This work deals entirely with safety properties; future work could consider performance and fault-tolerance properties using the conditional performance analysis as presented in [17]. 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.

In a related work [11] we also have investigated a generalization of the DVS service to dynamic *sets of primaries* rather than individual primaries, in order to tolerate 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 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.

[4] 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.

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

[6] K.P. Birman, *A Review of Experiences with Reliable Multicast*, Software– Practice and Experience, (J. Wiley), vol. 29, no. 9, pp. 741-774, Aug. 1999.

[7] 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.

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

[9] 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.

[10] 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.

[11] R. De Prisco, A. Fekete, N. Lynch, A. Shvartsman, "A dynamic primary configuration group communication service", in Proceedings of the 13^{th} International Symposium of Distributed Computing (DISC 99), Bratislava, Slovak.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] A. Fekete, N. Lynch and A. Shvartsman "Specifying and using a partitionable group communication service", *ACM Transaction on Computer Systems*, vol. 19, no. 2, pp. 171–216, May, 2001.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] Roger Khazan. "Group communication as a base for a load-balancing replicated data service", Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1998.

[24] Roger Khazan, Alan Fekete, and Nancy Lynch. "Multicast group communication as a base for a load-balancing replicated data service", In *12th International Symposium on Distributed Computing*, pages 258–272, Andros, Greece, September 1998.

[25] N. Lesley and A. Fekete, "Providing View Synchrony for Group Communication Services", *Proceedings of the Australian Computer Science Conference.* Auckland, New Zealand, January 1999, pp 457-468.

[26] 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.

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

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

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

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

[37] R. Vitenberg, I. Keidar, G.V. Chockler and D. Dolev, "Group Communication Specifications: A Comprehensive Study", MIT Technical Report MIT-LCS-TR-790, September 1999. URL `http://theory.lcs.mit.edu/~idish/ftp/gcs-survey-tr.ps`.