A Client-Server Approach to Virtually Synchronous Group Multicast: Specifications, Algorithms, and Proofs

Idit Keidar Roger Khazan

MIT Lab for Computer Science
Email: {idish, roger}@theory.lcs.mit.edu
URL: http://theory.lcs.mit.edu/{~idish, ~roger}

Abstract

This paper presents a formal design for a novel group multicast service that provides virtually synchronous semantics in asynchronous fault-prone environments. The design employs a client-server architecture in which group membership is maintained not by every process but only by dedicated membership servers, while virtually synchronous group multicast is implemented by service end-points running at the clients. This architecture allows the service to be scalable in the topology it spans, in the number of groups, and in the number of clients. Our design allows the virtual synchrony algorithm to run in a single message exchange round, in parallel with the membership algorithm: it does not require pre-agreement upon a common identifier by the membership algorithm.

Specifically, the paper defines service semantics for the client-server interface, that is, for the group membership service. The paper then specifies virtually synchronous semantics for the new group multicast service, as a collection of safety and liveness properties. These properties have been previously suggested and have been shown to be useful for distributed applications. The paper then presents new algorithms that use the defined group membership service to implement the specified properties. The specifications and algorithms are presented incrementally, using a novel *inheritance*-based formal construct [26]. The algorithm that provides the complete virtually synchronous semantics executes in a single message round, and is therefore more efficient than previously suggested algorithms providing such semantics. The algorithm has been implemented in C++. All the specifications and algorithms are presented using the I/O automaton formalism. Furthermore, the paper includes formal proofs showing that the algorithms meet their specifications. Safety properties are proven using invariant assertions and simulations. Liveness is proven using invariant assertions and careful operational arguments.

Keywords: Group Communication, Virtual Synchrony, Reliable Multicast, Formal Modeling

1 Introduction

Group communication systems [1, 39] are powerful building blocks that facilitate the development of fault-tolerant distributed applications. Group communication provides the notion of *group abstraction*, which allows processes to be easily organized in multicast groups. Group communication systems typically integrate two types of services: group membership and reliable group multicast. The membership service maintains a listing of the currently active and connected group members and delivers this information to its clients whenever it changes. The output of the membership service is called a *view*. Reliable multicast services that deliver messages to the current view members complement the membership service. For simplicity's sake, in this paper we restrict our attention to a single multicast group.

Group communication systems usually run in asynchronous fault-prone environments. In such environments, group communication systems generally provide some variant of *virtual synchrony* semantics which synchronize membership notifications with regular messages and thus simulate a "benign" world in which message delivery is reliable within the set of connected processes. Such semantics are especially useful for constructing fault-tolerant applications that maintain consistent replicated state of some sort (e.g., [3, 6, 25, 18, 37, 9, 28]). The key aspect of virtual synchrony is the semantics of interleaving of message send and delivery events with view delivery events. In order to reason about this interleaving, we associate message send and delivery events with views: we say that an event e occurs at a process p *in view* $\mathtt{v}$ if $\mathtt{v}$ was the last view delivered to p before e, or a default initial view $\mathtt{v}_{\mathtt{p}}$ if no such view was delivered.

Many variants of virtual synchrony semantics have been suggested [32, 19, 39, 12, 34, 18]. All of these variants specify that every message is delivered in the same view by all processes that deliver it. Some of these semantics (e.g., *strong virtual synchrony* [19]) strengthen this property to require that the view in which a message is delivered is the same view in which it was sent. Another useful property specified by nearly all variants of virtual synchrony is the agreement of the processes moving together from a view $\mathtt{v}$ to another view $\mathtt{v}'$ on the set of messages delivered in $\mathtt{v}$. Our group communication service specification includes all of these, as well as several additional safety and liveness properties. In Section 4 we present the properties we have chosen to provide.

Traditionally, virtual synchrony semantics were implemented by algorithms that were integrated with group membership algorithms (e.g., in [19, 22, 7]). Recently, Keidar et al. [27, 8] proposed a novel client-server oriented group membership service that decouples membership maintenance from group multicast, in order to provide scalable membership services in a wide area network (WAN). In their approach, a small set of dedicated membership servers maintains client membership information (i.e., which clients are members of each group). Virtual synchrony is achieved by service end-points running at the clients. This architecture allows the group communication system to be scalable in the topology it spans, in the number of groups and in the number of clients.

In this paper we present a novel client-server oriented virtually synchronous group multicast service that interacts with such an *external* membership service. Introducing the client-server design poses a major challenge: One has to define an interface by which a membership server interacts with its clients, in a way that would allow for simple and efficient implementations of both group membership (by the membership servers), and virtual synchrony (by service end-points at the clients). Such an interface has to provide sufficient level of synchronization to allow the virtual synchrony algorithm to reach agreement upon the set of messages delivered in the old view in *parallel* with the servers' agreement on views. In addition, one has to try to minimize the communication overhead induced due to messages sent as part of this interface.

We have designed an interface that addresses the challenges listed above. Our interface consists of two types of messages sent from servers to their clients: When a server engages in a view change algorithm, it sends its clients a `start_change` message notifying them that a view change is in progress. Each `start_change` message contains a locally (per-client) unique identifier. This identifier is *not* globally agreed upon: `start_change` messages sent to different processes can contain different identifiers. Once the server agrees upon the new view with the other servers, it notifies the clients of the view via a `view` message. The view contains information that maps processes to the last `start_change` identifiers they received before receiving this view. The servers do not need to receive messages from all their clients in order to complete the view change algorithm. The client-server interface is presented in Section 3.

Our interface allows for straightforward implementations of both membership and virtual synchrony: In Section 5 we present a simple one-round virtual synchrony algorithm that exploits this interface. We have implemented this algorithm in C++ using the scalable membership service of [27], which is a simple one-round membership algorithm providing the specified interface and assumptions. With our algorithm, the virtual synchrony round and the membership round are conducted in parallel: once the clients receive the `start_change` notifications, clients send each other special synchronization messages which allow them to agree upon the set of messages to be delivered before moving to the new view. We are not aware of any other algorithm that implements virtual synchrony in one communication round without pre-agreement upon a globally unique identifier by the membership algorithm.

During the period in which the group communication service is attempting to reach agreement on a view, new processes may attempt to join. In such cases, previously suggested virtual synchrony algorithms, e.g., [22, 16], can have the current invocation of the membership and virtual synchrony algorithms proceed to termination without adding the new processes, and then invoke the algorithms again to add the new processes. In contrast, our algorithm never delivers views that reflect a membership that is already known to be out of date. Thus, we avoid inducing extra overhead for applications to process unnecessary views. In addition, our algorithm allows some application messages to be delivered while it is reconfiguring.

Throughout this paper we use the I/O automaton formalism (cf. [31] and [30], Ch. 8) to provide rigorous specifications and algorithm descriptions. Previously suggested I/O automaton-style specifications of group communication systems used a single abstract automaton to represent multiple properties of the same system component and presented a single algorithm automaton that implements all of these properties. Thus, no means were provided for reasoning about a subset of the properties, and it was often difficult to follow which part of the algorithm implements which part of the specification. We address this shortcoming by specifying separate properties as separate abstract automata, and by incrementally constructing the algorithm that implements them – in each step adding support for an additional property – using a novel *inheritance*-based formal construct, recently introduced to the I/O automaton model [26].

We formally prove the correctness of the presented algorithms, that is, that they meet the specifications of Section 4. In Section 6, we prove that the safety properties of our algorithms are satisfied using invariant assertions and simulations. The safety proof is *modular*: we exploit the inheritance-based structure of our specifications and algorithms to reuse proofs. In Section 7, we prove the liveness of our algorithm using invariant assertions and operational arguments.

2 Formal Model and Notation

In the I/O automaton model (cf. [31] and [30], Ch. 8), a system component is described as a state-machine, called an *I/O automaton*. The transitions of this state-machine are associated with named actions, which are classified as either *input*, *output*, or *internal.* Input and output actions model the component's interaction with other components, while internal actions are externally-unobservable.

Formally, an I/O automaton is defined as the following five-tuple: a signature (input, output and internal actions), a set of states, a set of start states, a state-transition relation (a cross-product between states, actions, and states), and a partition of output and internal actions into *tasks.* Tasks are used for defining fairness conditions on an execution of the automaton.

An action π is said to be *enabled* in a state `s` if the automaton has a transition of the form (`s`, π, `s'`); input actions are enabled in every state. An *execution* of an automaton is an alternating sequence of states and actions that begins with its start state and in which every action is enabled in the preceding state. An infinite execution is *fair* if, for each task, it either contains infinitely many actions from this task or infinitely many occurrences of states in which no action from this task is enabled; a finite execution is *fair* if no action is enabled in its final state. A *trace* is a subsequence of an execution consisting solely of the automaton's external actions. A *fair trace* is a trace of a fair execution.

When reasoning about an automaton, we are only interested in its externally-observable behavior as reflected in its traces. There are two types of trace properties: *safety* and *liveness.* Safety properties usually specify that some particular bad thing never happens. In this paper we specify safety properties using (abstract) I/O automata that generate the legal sets of traces; for such automata we do not specify task partitions. An implementation automaton satisfies a specification if all of its traces are also traces of the specification automaton. Refinement mappings are commonly used technique for proving trace inclusion, in which one automaton (the algorithm) *simulates* the behavior of another automaton (the specification). Refinement mappings and other related techniques are reviewed in Appendix A. Liveness properties usually specify that some good thing eventually happens. An implementation automaton satisfies a liveness property if the property holds in all of its *fair* traces.

The *composition operation* defines how automata interact via their input and output actions: It matches output and input actions with the same name in different component automata; when a component automaton performs a step involving an output action, so do all components that have this action as an input one. The result of composing an output action with an input action is classified as an output to allow for future compositions with other automata. A *hide* operator can re-classify an output action as an internal one to prevent it from being externally observable. When reasoning about a certain system component, we compose it with abstract specification automata that specify the behavior of its environment.

I/O automata are conveniently presented using the *precondition-effect* style: In this style, typed state variables with initial values specify the set of states and the start states. A variable type is a set (if S is a set, the notation $S_\perp$ refers to the set $S \cup \{\perp\}$). Transitions are grouped by action name, and are specified as a list of triples consisting of an action name (possibly with parameters), a `pre:` block with preconditions on the states in which the action is enabled and an `eff:` block which specifies how the pre-state is modified; the effect is executed *atomically* to yield the post-state.

We use a novel *inheritance*-based formal concept, recently introduced into the I/O automaton model [26]. A *child* automaton is specified as a modification of the parent automaton's code, which restricts the parent's behavior. When presenting a child we first specify a *signature extension* which consists of new actions (labeled new) and modified actions (the modified action is labeled with the name of the action which they modify as follows: modifies `parent.action(parameters)`). We next specify the *state extension* consisting of new state variables added by the child. Finally, we describe the *transition restriction* which consists of new preconditions and effects added by the child to both new and modified actions. For modified actions, the preconditions and effects of the parent are appended to those added by the child. New effects added by the child are performed before the effects of the parent, all of them in a single atomic step. The child's effects are not allowed to modify state variables of the parent. This restriction ensures that the set of traces of the child, when projected onto the parent's signature, is a subset of the parent's set of traces.

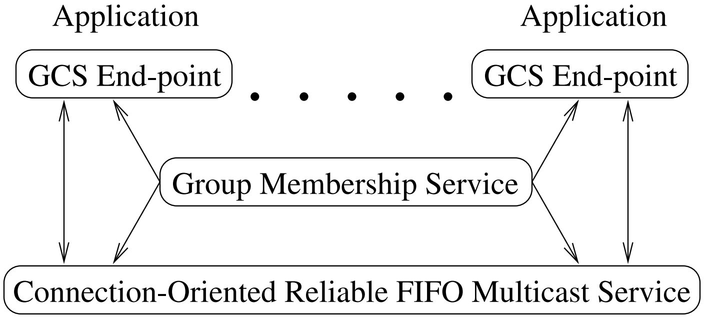
Inheritance allows us to reuse code and avoid redundancies. It also allows us to reuse proofs: Assume that an implementation automaton `I` can simulates a specification automaton `S`, and let `I'` and `S'` be child automata of `I` and `S`, respectively. Then the Proof Extension theorem of [26] asserts that in order to prove that `I'` can simulate `S'` it is sufficient to show that the restrictions added by `I'` are consistent with the restrictions `S'` places on `S`, and that the new functionality of `I'` can simulate new functionality of `S'`. Appendix A contains more detailed information on the Proof Extension theorem of [26].

3 Client-Server Architecture and Environment Specification

Our service is implemented in an asynchronous message-passing environment. Processes can crash and recover; communication links may fail and may later recover. Initially, we assume that stable storage is used, that is, we assume that crashed processes recover with their state intact. We do not explicitly model process crashes as under this asusmption, a process crash is equivalent to that processes' migration into a singleton view. In Section 8 we show that our algorithms also provide meaningful semantics without using stable storage, and explain how crash and recovery events can be modeled.

The service is implemented by *group communication service end-points (GCS end-points)* running at client processes. Throughout this paper we use the words "process" and "end-point" interchangeably. The end-points receive views from an external membership service whose specification appears in Section 3.1. The group communication end-points communicate with each other using a reliable FIFO multicast service which we describe in Section 3.2. We specify the membership and multicast services using centralized (global) I/O automata. Each external action in these specifications is tagged with a subscript `p`, which denotes the process at which this action occurs. The architecture is depicted in Figure 1.

Figure 1 The client-server architecture: GCS end-points using an external membership service.



3.1 The membership service specification

In Figure 2 we specify an external membership service whose interface consists of two output events: Like every other membership service, the service provides its clients with *views* that contain a set of members currently believed to be alive and an increasing identifier. In addition, it provides `start_change` actions which notify clients that the membership service is attempting to form a new view. The membership service is *partitionable* [16, 39, 10], i.e., allows several disjoint views to exist concurrently. It can be implemented by dedicated membership servers. The MBRSHP automaton interface consists of two types of output actions:

Figure 2 Membership service safety specification.

```
AUTOMATON MBRSHP

Type View: ViewId x SetOf(Proc) x (Proc → StartChangeId)

Signature:
 Output: start_change_p(cid, set), Proc p, StartChangeId cid, SetOf(Proc) set
         view_p(v), Proc p, View v

State:
 Forall Proc p: View  mbrshp_view[p], initially v_p = ⟨vid_0, {p}, {(p →cid_0)}⟩
 Forall Proc p:  (StartChangeId x SetOf(Proc))  start_change[p], initially ⟨cid_0, {}⟩
 Forall Proc p: mode[p] ∈ {normal, change_started}, initially normal

Transitions:
 OUTPUT  start_change_p(cid, set)
 pre: cid > start_change[p].id
      p ∈ set
 eff: start_change[p] ← ⟨cid, set⟩
      mode[p] ← change_started

 OUTPUT  view_p(v)
 pre: v.id > mbrshp_view[p].id
      v.set ⊆ start_change[p].set ∧ p ∈ v.set
      v.startId(p) = start_change[p].id
      mode[p] = change_started
 eff: mbrshp_view[p] ← v
      mode[p] ← normal
```

$\texttt{start\_change}_\texttt{p}(\texttt{cid}, \texttt{set})$ notifies process `p` that the membership service is attempting to form a new view with the members in `set`. `cid` is a locally unique (increasing) start_change identifier (taken from a totally ordered set `StartChangeId` with the smallest element $\texttt{cid}_0$, e.g., integers).

$\texttt{view}_\texttt{p}(\texttt{v})$ notifies process `p` of the new view `v`. A view `v` is a triple consisting of an increasing identifier `v.id` (from a partially ordered set `ViewId` with the smallest element $\texttt{vid}_0$, e.g., integers), a set of members `v.set` (which is a subset of the `set` in the latest preceding $\texttt{start\_change}_\texttt{p}(\texttt{cid}, \texttt{set})$), and a function `v.startId` that maps members of `v` to the `cid`s in the last start_changes they received before receiving `v`. Two views are considered to be the same if they consist of identical triples.

The membership specification captures two basic membership properties: *Self Inclusion* requires every view delivered to an end-point `p` to include `p` as a member, and *Local Monotonicity* requires that view identifiers delivered to an end-point be monotonically increasing. Local Monotonicity has two important consequences: it guarantees that the same view is not delivered more then once to the same end-point and that if two views are delivered to two end-points they are delivered in the same order. These properties are fulfilled by virtually all group membership services (e.g., [12, 16, 7, 19, 10, 27, 34, 5]).

In addition, using the `mode[p]` variable, the MBRSHP automaton requires that the membership service send at least one `start_change` to an end-point `p` prior to every view `v` sent to `p`. It also requires that the start_change identifier `v.startId(p)` be the same as the `cid` of the latest `start_change` delivered to `p` before the view, and that `v.set` be a subset of the set suggested in that `start_change`. Note that this specification does allow the membership service to add new processes while it is reconfiguring, as long as a new `start_change` is sent to the clients.

Keidar et al. [27] describe a simple and efficient service that satisfies this specification; the `start_change` interface was explicitly built into [27]. It would be straightforward to extend other membership algorithms (e.g., [16, 7]) to provide this interface, by having these algorithms send `start_change` messages to processes when they start the membership algorithm, and then include the start change identifiers in the view.

The specified service allows its clients to implement virtual synchrony as follows: When a process receives a `start_change`, it begins engaging in the virtual synchrony algorithm and sends synchronization messages to its peers. The process then waits to receive such messages from its peers, and uses them to decide upon the set of messages that will be delivered in the old view before moving to the new one. In order to have processes *agree* upon the set of messages in the old view, it is necessary that they all use the same set of synchronization messages. In order to uniquely identify this set of messages, previously suggested virtual synchrony algorithms (e.g., [7, 22]) had the processes first agree upon a common identifier (usually the view identifier of the view to be delivered), and then exchange synchronization messages tagged with this identifier. In contrast, our algorithm tags synchronization messages with start_change identifiers which are not globally agreed upon, and includes in each view a function `startId` which maps each process to the start change identifiers provided to it. Thus, the specification allows the virtual synchrony algorithm to execute in parallel with the membership algorithm without requiring pre-agreement on a globally unique identifier.

We do not specify any liveness property for the membership service. Instead, we state and prove the liveness properties of our group multicast service conditionally on the behavior of the membership service. The membership service of [27] which we use in our implementation satisfies a liveness guarantee which is, in turn, conditional on the behavior of the underlying network and of the failure detector it employs.

3.2 The connection-oriented reliable FIFO multicast service specification

The group communication end-points communicate with each other using an underlying connection-oriented reliable FIFO multicast service; this service provides reliable FIFO communication between every pair of connected processes. Many existing group communication systems (e.g., [22, 10, 16, 5]) implement virtual synchrony over similar underlying reliable communication substrates. In our implementation, we currently use the service of [36].

We specify the reliable FIFO multicast service in Figure 3 as a centralized automaton CO_RFIFO. CO_RFIFO maintains a FIFO queue `channel[p][q]` for every pair of end-points. The input action $\mathtt{send}_\mathtt{p}(\mathtt{set},\mathtt{m})$ models the multicast of message `m` from end-point `p` to the end-points listed in the `set` by appending it to the queues `channel[p][q]` for every end-point `q` in `set`. The $\mathtt{deliver}_{\mathtt{p},\mathtt{q}}(\mathtt{m})$ action removes the first message from `channel[p][q]` and delivers it to the end-point `q`.

An end-point `p` uses the action $\mathtt{reliable}_\mathtt{p}(\mathtt{set})$ to inform CO_RFIFO that it wishes to maintain reliable (gap-free) FIFO connections to the end-points in `set`, and to them only. As an effect of this action, `set` is stored in the variable `reliable_set[p]`. For every process `q` not in `reliable_set[p]`, CO_RFIFO may lose an arbitrary suffix of the messages sent from `p` to `q`, as modeled by the action `lose(p,q)`.

When specifying the liveness of CO_RFIFO, we would like to require that messages sent to live and connected processes eventually reach their destinations. Unfortunately, we cannot use `reliable_set[p]` for this purpose, as it does not necessarily reflect the *real* network situation because the `reliable_set[p]` is controlled by the client. Therefore, we use an additional variable, `live_set[p]`, which is assumed to reflect the real state of the network, that is, the set of processes

Figure 3 Connection-oriented reliable FIFO multicast service specification.

```
AUTOMATON CO_RFIFO

Signature:
 Input:    send_p(set, m), Proc p, SetOf(Proc) set, Msg m
           reliable_p(set), Proc p, SetOf(Proc) set
           live_p(set), Proc p, SetOf(Proc) set
 Output:   deliver_p,q(m), Proc p, Proc q, Msg m
 Internal: lose(p,q), Proc p, Proc q

State:
 For all Proc p, Proc q: SeqOf(Msg) channel[p][q], initially empty
 For all Proc p:  SetOf(Proc) reliable_set[p], initially {p}
 For all Proc p:  SetOf(Proc) live_set[p], initially {p}

Transitions:
 INPUT  send_p(set, m)
 eff: (∀ q ∈ set) append m to channel[p][q]

 OUTPUT  deliver_p,q(m) hidden param live_set[p]
 pre: m = First(channel[p][q])
 eff: dequeue m from channel[p][q]

 INPUT  reliable_p(set)
 eff: reliable_set[p] ← set

 INTERNAL  lose(p, q)
 pre: q ∉ reliable_set[p]
 eff: dequeue last message from channel[p][q]

 INPUT  live_p(set)
 eff: live_set[p] ← set

Tasks:
 (∀ p)(∀ q ∈ live_set[p]) {deliver_p,q(m)}
 {dummy()} ∪ {deliver_p,q(m)  |  q ∉ live_set[p]} ∪ {lose(p,q)}
```

which are really alive and connected to p. This variable is set by the action $\texttt{live}_\texttt{p}(\texttt{set})$, and is used in the definition of CO_RFIFO's tasks to require that a message sent from p to a process in live_set[p] be delivered at some point.

4 Specifications of the Group Communication Service

Our group communication service satisfies a set of common properties previously suggested in several variants of virtual synchrony. These properties have been proven to be useful for many distributed applications. In Section 4.1, we specify the safety properties satisfied by our service as abstract I/O automata (without tasks). We present the liveness property of the group communication service in Section 4.2.

4.1 Safety properties

The safety properties of our service are specified as centralized (global) I/O automata. Each action in the specifications is tagged with a subscript p, which denotes the process at which this action occurs. We present our specifications in four steps, as four automata: In Section 4.1.1 we specify a simple group communication service that provides reliable FIFO multicast within views. In Section 4.1.2 we extend the specification of Section 4.1.1, to require also that processes moving together from view v to view v' deliver the same set of messages in view v. In Section 4.1.3 we specify a service which provides *transitional sets* (first presented as part of *Extended Virtual Synchrony (EVS)* [32]). In Section 4.1.4 we specify the Self Delivery property which requires processes to deliver their own messages. We note that our specifications are partitionable [16, 39, 10] in that they allow several disjoint views to exist concurrently.

4.1.1 Within-view reliable FIFO multicast

In Figure 4 we present the within-view reliable FIFO (WV_RFIFO) service specification. The specification uses centralized queues of application messages, msgs[p][v], for each sender p for each view v. The action $\mathtt{send}_p(\mathtt{m})$ models the multicast of message m from process p to the members of p's current view by appending m to msgs[p][current_view[p]]. The $\mathtt{deliver}_p(\mathtt{q,m})$ action models the delivery to process p of message m sent by process q while in p's current view. The specification enforces gap-free FIFO ordered delivery of messages by using the variable last_dlvrd[q][p] to index the last message from q delivered to p in p's current view.

Figure 4 Within-view reliable FIFO multicast service specification.

```
AUTOMATON WV_RFIFO : SPEC

Signature:
 Input:  send_p(m), Proc p, AppMsg m          Output: deliver_p(q, m), Proc p, Proc q, AppMsg m
                                                      view_p(v), Proc p, View v

State:
 For each Proc p, View v: SeqOf(AppMsg) msgs[p][v], initially empty
 For each Proc p, Proc q: Int last_dlvrd[p][q], initially 0
 For each Proc p:         View current_view[p], initially v_p

Transitions:
 INPUT  send_p(m)
 eff: append m to msgs[p][current_view[p]]

 OUTPUT  deliver_p(q, m)
 pre: m = msgs[q][current_view[p]] [last_dlvrd[q][p]+1]
 eff: last_dlvrd[q][p] ← last_dlvrd[q][p]+1

 OUTPUT  view_p(v)
 pre: p ∈ v.set
      v.id > current_view[p].id
 eff: (∀ q) last_dlvrd[q][p] ← 0
      current_view[p] ← v
```

WV_RFIFO satisfies the following properties:

- It delivers membership views to the application in a manner that preserves the basic properties of the membership service: Local Monotonicity and Self Inclusion.
- WV_RFIFO delivers messages in the view in which they were sent. This property is useful for applications (cf. [19, 39, 37]) and appears in several systems (e.g., Isis [12], Horus [38] and Totem [7]) and specifications (e.g., [32, 18, 24, 15]). Some systems weaken this property by requiring that each message be delivered in the same view at every process that delivers it, but not necessarily the view in which it was sent. This weaker property is typically implemented on top of the implementation that provides within-view delivery.
- WV_RFIFO guarantees that messages are delivered in gap-free FIFO order (within views). We have chosen to provide FIFO multicast as opposed to a service with stronger ordering guarantees (causally or totally ordered), since FIFO is a basic service upon which one can build stronger services. For example, the totally ordered multicast algorithm of [13] is implemented atop a service that satisfies the WV_RFIFO specification.

4.1.2 Virtual synchrony

In Figure 5 we specify a virtually synchronous reliable FIFO multicast service, VS_RFIFO. We use inheritance to extend the WV_RFIFO specification presented above: VS_RFIFO is a child of the WV_RFIFO automaton. Thus, Figure 5 contains only the modification of the WV_RFIFO automaton, and the VS_RFIFO specification consists of the code presented in both Figures 4 and 5.

Like most variants of virtual synchrony (e.g., [32, 19, 39, 12, 34, 23]), our specification requires that processes moving together from view `v` to view `v'` deliver the same set of messages in `v`. This property by itself is often called *Virtual Synchrony*. In order to model this property, we add an internal action `set_cut(v, v', c)`, which non-deterministically fixes the set of messages to be delivered in view `v` by every process that moves from `v` to `v'`. The reliable FIFO order of message delivery allows us to represent the set of delivered messages using the index of the last delivered message from each sender.

Figure 5 Virtually synchronous reliable FIFO multicast service specification.

AUTOMATON VS_RFIFO : SPEC   MODIFIES WV_RFIFO : SPEC

Signature Extension:

```
Output:   view_p(v) modifies wv_rfifo.view_p(v)
Internal: set_cut(v, v', c), View v, View v', (Proc → Int)⊥ c new
```

State Extension:

```
For each View v, v': (Proc → Int)⊥ cut[v][v'], initially ⊥
```

Transition Restriction:

```
OUTPUT view_p(v)
pre: cut[current_view[p]] [v] ≠ ⊥
   (∀ q) last_dlvrd[q][p] = cut[current_view[p]] [v](q)

INTERNAL set_cut(v, v', c)
pre: cut[v][v'] = ⊥
eff: cut[v][v'] ← c
```

Note that the specification does not prevent processes from delivering messages beyond an established cut. However, if they do, they are not allowed to move into the view associated with this cut.

Virtual Synchrony is especially useful for applications that implement data replication using the state machine approach [35], (e.g., [37, 20, 4, 2, 25, 28]). When a new view forms, such applications must exchange special messages in order to synchronize members of the new view. A group communication system that supports Virtual Synchrony allows processes to avoid such costly exchange among processes that continue together from one view to the next.

4.1.3 Transitional set

While Virtual Synchrony is a useful property, a process that moves from view `v` to view `v'` cannot locally tell which of the processes in $\mathtt{v.set} \cap \mathtt{v'.set}$ move to view `v'` directly from view `v`, and which move to `v'` from some other view. In order for the application to be able to exploit the Virtual Synchrony property, application processes need to be told which other processes move together with them from their old views in to their new views. The set of such processes is called a *transitional set*. The notion of a transitional set was first introduced as part of a special, transitional view in the EVS [32] model. In our formulation, transitional sets are delivered to the applications together with (regular) views, as an additional parameter `T`. The delivery of transitional sets satisfies the following property (defined in [39]):

Property 4.1 (Transitional Set) *When a process* p *moves from view* v *to view* v′*, the transitional set it delivers with* v′ *is a subset of* v.set ∩ v′.set *which includes all the processes that move directly from* v *to* v′*, (including* p*), and does not include any member of* v′.set *that moves to* v′ *from any view other than* v*.*

Note that different transitional sets may be associated with the same view v′ at different processes (if they move to v′ from different previous views).

Figure 6 Transitional set specification.

```
AUTOMATON TRANS_SET : SPEC

Signature:
 Output:   view_p(v, T), Proc p, View v, SetOf(Proc) T
 Internal: set_prev_view_p(v), Proc p, View v

State:
 For each Proc p:  View current_view[p], initially v_p
 For each Proc p, View v: View⊥ prev_view[p][v], initially ⊥

Transitions:
 OUTPUT  view_p(v, T)
 pre: prev_view[p][v] = current_view[p]
      (∀ q ∈ v.set ∩ current_view[p].set)
            prev_view[q][v] ≠ ⊥
      T = {q ∈ v.set ∩ current_view[p].set |
            prev_view[q][v] = current_view[p]}
 eff: current_view[p] ← v

 INTERNAL  set_prev_view_p(v)
 pre: p ∈ v.set
      prev_view[p][v] = ⊥
 eff: prev_view[p][v] ← current_view[p]
```

We present the transitional set specification in Figure 6 as a separate automaton (without using inheritance). In order to compute the transitional set for a view v′, a process needs to know the previous views of the other processes that move to view v′. We use the variable prev_view to keep track of these previous views as follows: A process "declares" its intention to move to v′ from the current view by means of the action set_prev_view(v′) which sets prev_view[p][v′] to the current view. Before a process p can move to view v′ from view v, all the members of v.set ∩ v′.set must call set_prev_view(v′) to "declare" from which view they intend to move to v′. The transitional set of p at the time of delivering view v′ is then computed to consist of those processes q in v.set ∩ v′.set for which prev_view[q][v′] = v.

4.1.4 Self delivery

In Figure 7 we specify the Self Delivery property as a modification to the WV_RFIFO specification automaton which requires that an end-point p not deliver a new view without having delivered all the messages the application at p had sent in its current view. Note that we specify Self Delivery as a safety property. In contrast, other specifications (e.g., [39, 32]) formulate Self Delivery as a liveness property which requires processes to eventually deliver their own messages. Together with the liveness property we state below, Property 4.2, our specification of Self Delivery implies the Self Delivery liveness property stated in [39].

In order to implement Self Delivery together with Virtual Synchrony in a live manner, applications of the service must be *blocked* from sending new messages while a view change is taking place (as proven in [19]).

Figure 7 Self Delivery property specification.

AUTOMATON SELF : SPEC   MODIFIES WV_RFIFO : SPEC

Signature Extension:
Output: $\mathtt{view_p(v)}$ modifies $\mathtt{wv\_rfifo.view_p(v)}$

Transition Restriction:
OUTPUT **$\mathtt{view_p(v)}$**
pre: last_dlvrd[p][p] = LastIndexOf(msgs[p][current_view[p]])

4.2 Liveness property

We now specify liveness of our group communication system. Liveness is an important complement to safety, since without requiring liveness, safety properties can be satisfied by trivial implementations which do nothing. Liveness of a group communication service is bound to depend on the behavior of the underlying network. In a fault-prone asynchronous model, it is not feasible to require that the group communication service be live in every execution. The only way to specify useful liveness properties without strengthening the communication model is to make these properties *conditional* on the underlying network behavior (as specified, e.g., in [18, 14, 39]).

Since in this paper we use an external membership service, our liveness property is conditional on the behavior of the membership service. This approach is meaningful when we use an external membership service which itself satisfies some meaningful liveness properties (e.g.,the service of [27]).

Our liveness specification requires that if the membership eventually *stabilizes*, i.e., delivers the same view to all the view members and does not deliver any subsequent views, then the group communication service also delivers this view to all of its applications and delivers all the messages sent in this view. Formally:

Property 4.2 (Liveness) *Let* v *be a view with* $\mathtt{v.set} = \mathtt{S}$*. Let* α *be a fair execution sequence of a group communication service* GCS *in which, for every* $\mathtt{p} \in \mathtt{S}$*, the action* MBRSHP.$\mathtt{view_p(v)}$ *occurs and is followed by neither* MBRSHP.$\mathtt{view_p}$ *nor* MBRSHP.$\mathtt{start\_change_p}$ *actions. Then at each end-point* $\mathtt{p} \in \mathtt{S}$*,* GCS.$\mathtt{view_p(v)}$ *eventually occurs. Furthermore, for every* GCS.$\mathtt{send_p(m)}$ *that occurs after* GCS.$\mathtt{view_p(v)}$*, and for every* $\mathtt{q} \in \mathtt{S}$*,* GCS.$\mathtt{deliver_q(p,m)}$ *also occurs.*

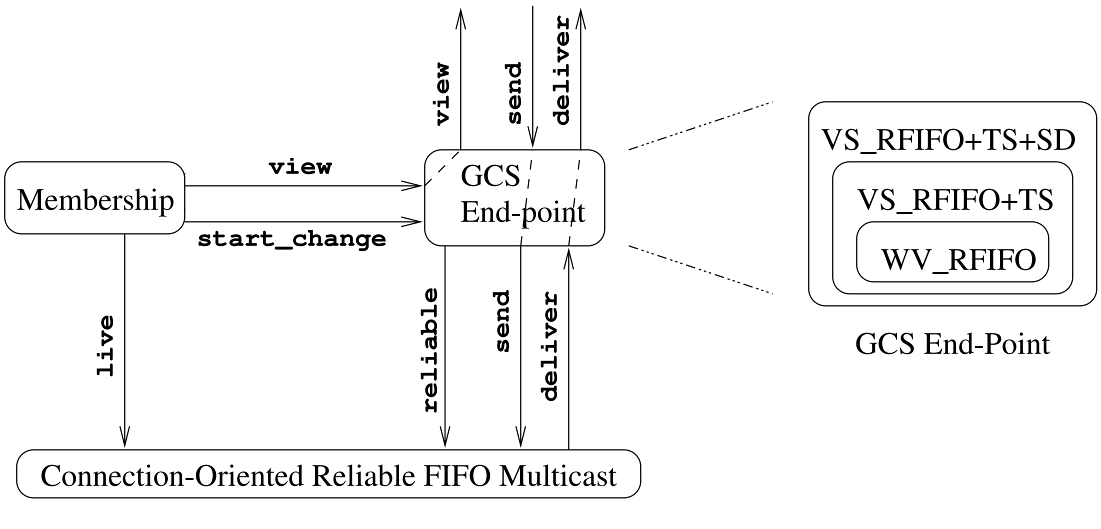
It is important to note that although our liveness property requires a GCS to be live only in certain executions, the conditions on these executions are *external* to the GCS implementation. Thus, any implementation which satisfies our liveness requirement has to attempt to be live in every execution because it cannot know whether or not the membership has permanently stabilized.

Note also that formally, membership stability is required to last forever. In practice, it only has to hold "long enough" for the group communication service to reconfigure, as explained in [17, 21]. However in an asynchronous model, we cannot explicitly introduce the bound on this time period, because its duration depends on external conditions such as message latency, process scheduling and processing time.

5 The Virtually Synchronous Group Multicast Algorithm

The group communication service is implemented by symmetric GCS end-points, i.e., all end-points run the same algorithm. When reasoning about our algorithm, we consider a composition of all end-point automata with the automata specifying their environment, MBRSHP and CO_RFIFO, described in Section 3. After composing the automata, we hide all the output actions except for the application interface.

Figure 8 A GCS end-point and its environment.



(a) Interaction with Membership and Multicast Services **(b)** Inheritance Hierarchy

Figure 8 (a) shows the interaction of a GCS end-point with its environment. The application sends messages via the GCS end-point, and the end-point delivers application messages and views to the application. The GCS, in turn, uses CO_RFIFO to send messages to other GCS end-points, and CO_RFIFO delivers such messages to the GCS. The GCS uses the action CO_RFIFO.$\mathtt{reliable}_p(\mathtt{set})$ to inform CO_RFIFO that it wishes to maintain reliable (gap-free) FIFO connections to the end-points in `set`, and to them only.

The GCS end-point also receives `start_change` and `view` notifications from the membership service. In addition, MBRSHP.$\mathtt{start\_change}_p(\mathtt{id}, \mathtt{set})$ is linked with CO_RFIFO.$\mathtt{live}_p(\mathtt{set})$, and MBRSHP.$\mathtt{view}_p(\mathtt{v})$ with CO_RFIFO.$\mathtt{live}_p(\mathtt{v.set})$. This way, the `live_set[p]` at CO_RFIFO matches the membership's perception of which processes are alive and connected to `p`. Linking the MBRSHP output actions with the CO_RFIFO`.live` inputs reflects our assumption that every permanently disconnected end-point is eventually excluded either from a `start_change` notification or from a `view`.

We construct the algorithm in steps, at each step adding support for a new property (see Figure 8 (b)):

- In Section 5.1, we present an algorithm WV_RFIFO_p for an end-point of a within-view reliable FIFO multicast service, and argue that the group communication system built from these end-points satisfies the safety specification WV_RFIFO : SPEC and the liveness Property 4.2.

- In Section 5.2, we modify the WV_RFIFO_p algorithm to also satisfy the Virtual Synchrony and Transitional Set properties. We present an algorithm VS_RFIFO+TS_p, which is a child of WV_RFIFO_p, which allows us to reuse the correctness of the within-view reliable FIFO service to argue that the group communication system built from VS_RFIFO+TS_p end-points satisfies the safety specifications VS_RFIFO : SPEC and TS : SPEC.

- In Section 5.3, we modify VS_RFIFO+TS_p to also satisfy the Self Delivery property. The resulting automaton VS_RFIFO+TS+SD models the algorithm of our group communication service GCS. Since we use inheritance, the final algorithm automatically satisfies WV_RFIFO : SPEC, VS_RFIFO : SPEC, and TS : SPEC safety specifications. We argue that it also satisfies SELF : SPEC and Property 4.2.

In the presented end-point automata, each locally controlled action is defined to be a task by itself, which means that it eventually happens if it becomes enabled and is not subsequently disabled by another action. In this section we present informal correctness arguments; formal correctness proofs appear in Sections 6 and 7.

We would like to note that the algorithm automata present the key ideas at a level that would be easy to follow. We supplement this presentation with a discussion of some important optimizations that a real implementation of our algorithm may perform.

5.1 Within-view reliable FIFO multicast algorithm

WV_RFIFO : SPEC (cf. Section 4.1.1) requires the following properties: first, it requires membership views to be delivered to the application in a manner that preserves Local Monotonicity and Self Inclusion; second, it requires that messages be delivered in the views in which they were sent; and finally it requires that messages be delivered in a gap-free FIFO order (within views).

WV_RFIFO forwards to its application views generated by the MBRSHP service which satisfies Local Monotonicity and Self Inclusion, and it multicasts messages using the CO_RFIFO service that provides gap-free FIFO communication. Thus, the only task left to WV_RFIFO is to synchronize message delivery with views in order to guarantee that messages be delivered in the views in which they were sent.

To this end, WV_RFIFO sends a `view_msg(v)` to all the members of a view `v` before sending them application messages in `v`. An application message from sender `q` received via CO_RFIFO is associated with the view conveyed by the latest `view_msg(v)` from `q`. WV_RFIFO delivers to its application only messages that are associated with its current view.

In addition, our WV_RFIFO service allows processes to forward application messages on behalf of other processes. Thus, it does not put itself at mercy of CO_RFIFO to determine which messages to deliver in a given view. WV_RFIFO does not establish a particular message forwarding strategy. Actual implementations and extensions of WV_RFIFO may define the forwarding strategy appropriate for them by adding preconditions on the action that sends forwarded messages.

In Figure 9 we present the WV_RFIFO_p automaton at end-point `p`. Views received from MBRSHP are stored in `mbrshp_view` and then delivered to the application. The variable `current_view` contains the latest view delivered to the application. WV_RFIFO satisfies Self-Inclusion and Local Monotonicity on views since WV_RFIFO forwards views generated by MBRSHP to its application without changing their order.

When an application sends a message, this message is appended to `msgs[p][current_view]`. Messages from `msgs[p][current_view]` are multicast in FIFO order to the other view members using CO_RFIFO.`send`. The index `last_sent` points to the last message in `msgs[p][current_view]` that was sent. Since `view_msgs` are sent within the message stream, end-points can associate application messages sent by their peers with the views in which they were sent: multicast messages are received using CO_RFIFO.$\texttt{deliver}_{q,p}$ and stored in `msgs[q][view_msg[q]]`, where `view_msg[q]` is the view in the last `view_msg` received from `q`. Messages are delivered to the application from `msgs[q][current_view]` (in FIFO order) so they are delivered in the view in which they were sent. `last_rcvd[q]` is the index of the last message received from `q`, and `last_dlvrd[q]` is the index of

Figure 9 Within-view reliable FIFO multicast end-point automaton.

```
AUTOMATON WV_RFIFO_p

Signature:
 Input:  send_p(m), AppMsg m
         co_rfifo.deliver_q,p(m), Proc q, (AppMsg + ViewMsg + FwdMsg) m
         mbrshp.view_p(v), View v

 Output: deliver_p(q, m), Proc q, AppMsg m
         co_rfifo.send_p(set, m), SetOf(Proc) set, (AppMsg + ViewMsg + FwdMsg) m
         view_p(v), View v

State:
 // Variables for handling application messages
 For all Proc q, View v: SeqOf(AppMsg_⊥) msgs[q][v], initially empty
 Int last_sent, initially 0
 For all Proc q: Int last_rcvd[q],  initially 0
 For all Proc q: Int last_dlvrd[q], initially 0

 // Variables for handling views and view messages
 View current_view, initially v_p
 View mbrshp_view,  initially v_p
 For all Proc q: View view_msg[q], initially v_q

 SetOf(Proc) reliable_set, initially v_p.set

Transitions:
 INPUT  mbrshp.view_p(v)
 eff: mbrshp_view ← v

 OUTPUT  view_p(v)
 pre: v = mbrshp_view
      v.id > current_view.id
 eff: current_view ← v
      last_sent  ← 0
      (∀ q) last_dlvrd[q] ← 0

 OUTPUT  co_rfifo.reliable_p(set)
 pre: current_view.set ⊆ set
 eff: reliable_set ← set

 OUTPUT  co_rfifo.send_p(set, tag=view_msg, v)
 pre: view_msg[p] ≠ current_view
      current_view.set ⊆ reliable_set
      set =  current_view.set - {p}
      v = current_view
 eff: view_msg[p] ← current_view

 INPUT  co_rfifo.deliver_q,p(tag=view_msg, v)
 eff: view_msg[q]  ← v
      last_rcvd[q] ← 0

 INPUT  send_p(m)
 eff: append m to msgs[p][current_view]

 OUTPUT  deliver_p(q, m)
 pre: m = msgs[q][current_view][last_dlvrd[q]+1]
      (q = p)  ⇒  (last_dlvrd[q] < last_sent)
 eff: last_dlvrd[q] ← last_dlvrd[q] + 1

 OUTPUT  co_rfifo.send_p(set, tag=app_msg, m)
 pre: view_msg[p] = current_view
      set =  current_view.set - {p}
      m = msgs[p][current_view][last_sent + 1]
 eff: last_sent ← last_sent + 1

 INPUT  co_rfifo.deliver_q,p(tag=app_msg, m)
 eff: msgs[q][view_msg[q]][last_rcvd[q]+1]←m
      last_rcvd[q] ← last_rcvd[q] + 1

 OUTPUT  co_rfifo.send_p(set,tag=fwd_msg,r,v,m,i)
 pre: m = msgs[r][v][i]

 INPUT  co_rfifo.deliver_q,p(tag=fwd_msg,r,v,m,i)
 eff:  msgs[r][v][i] ← m
```

the last message from q that was delivered to the application. Gap-free FIFO delivery of regular messages is guaranteed by the use of ordered message queues together with the guarantees of

CO_RFIFO. Note that an end-point cannot self-deliver its own message without first sending it to the other view members using CO_RFIFO.`send`.

Now, let us consider forwarded messages. Such messages are tagged with information about their original sender, the view in which they were originally sent, and their index in the `msgs` buffer. When a forwarded message is received, it is stored in the appropriate place in the appropriate buffer according to the index and view information it was tagged with. This information is correct since the original messages are associated with the correct information and forwarded messages directly or indirectly initiate from original messages.

Messages are always sent to all the current view members, since while messages are sent in a certain view, the `reliable_set` is a superset of that view. This is true since the action CO_RFIFO.$\mathtt{reliable_p(set)}$ is enabled only for a `set` which is a superset of `current_view.set`, and since a `view_msg` is not sent until the `reliable_set` has become a superset of `current_view.set`. In WV_RFIFO, the `reliable_set` is allowed to be an arbitrary superset of `current_view.set`. This set is further restricted in VS_RFIFO+TS.

WV_FIFO also satisfies the liveness Property 4.2: If a set of end-points receives the same view from the membership and no view events afterwards, then it is straightforward to show that, in a fair execution, each of these end-points eventually delivers this view to its application, as well as all messages sent subsequently in the context of this view. This relies on the liveness of CO_RFIFO, which guarantees that messages sent between connected processes (as perceived by the membership service) are eventually delivered.

Note that the algorithm as we have described it never removes messages from its buffers. Any actual implementation of the algorithm needs to employ some sort of a garbage collection mechanism for discarding old messages. Group communication systems usually use acknowledgments to track which messages have been delivered to all the view members, and such messages are discarded. In addition, whenever a process delivers a new view, it can discard messages from previous views.

5.2 Adding support for Virtual Synchrony and Transitional Sets

The WV_RFIFO service presented above guarantees that in each view `v` every member delivers some prefix of the FIFO ordered messages sent by each end-point in `v`. The VS_RFIFO+TS service presented in this section extends WV_RFIFO to guarantee also that those end-points which transition directly from view `v` to the same new view $\mathtt{v'}$ deliver not just "some" prefixes but "the same" prefixes of the FIFO ordered messages sent by each end-point in view `v` (cf. Property Virtual Synchrony in Section 4.1.2). Moreover, the VS_RFIFO+TS service delivers a transitional set, `T` with every view `v` such that `T` satisfies the Transitional Set property of Section 4.1.3. In order to satisfy these properties, an end-point moving from a view `v` to a view $\mathtt{v'}$ must first learn which other end-points may transition from `v` to $\mathtt{v'}$ and must agree with them on the lengths of the prefixes they need to deliver.

In a nutshell, here is how the VS_RFIFO+TS service accomplishes this: Each time an end-point is notified via MBRSHP.$\mathtt{start\_change_p(cid, set)}$ of the MBRSHP's attempt to form a new view, `p` reliably sends to `set` a synchronization message tagged with `cid`. This `cid` uniquely identifies the synchronization message, due to the local uniqueness of start_change identifiers. The synchronization messages are used for computing the transitional set and for agreeing upon the set of messages to be delivered to the application before moving to the next view. When MBRSHP.$\mathtt{view_p(v')}$ is delivered to `p`, `p` uses the mapping $\mathtt{v'.startId}$ in order to determine the set of synchronization messages to be used for these purposes: it uses a message tagged with $\mathtt{v'.startId(q)}$ from each end-point `q` in $\mathtt{v.set} \cap \mathtt{v'.set}$. Since $\mathtt{v'.startId}$ mapping is part of the view $\mathtt{v'}$, all the processes moving from view $\mathtt{v}$ to $\mathtt{v}'$ use the same set of synchronization messages for computing the transitional set and the set of messages to be delivered to the application before $\mathtt{v}'$. Thus, the inclusion of the `startId` mapping in views eliminates the need to pre-agree on a common tag for identifying which synchronization messages to consider for a given view.

5.2.1 Algorithm details and safety argument

Figure 10 presents the VS_RFIFO+$\mathrm{TS}_\mathtt{p}$ automaton as a child of WV_$\mathrm{RFIFO}_\mathtt{p}$. While there are no view changes, VS_RFIFO+$\mathrm{TS}_\mathtt{p}$ does not modify the behavior of WV_$\mathrm{RFIFO}_\mathtt{p}$. When a view change is taking place, VS_RFIFO+$\mathrm{TS}_\mathtt{p}$ sends and handles synchronization messages, and also restricts the delivery of application messages according to the synchronization messages associated with the new view.

Upon receiving a $\mathtt{start\_change}_\mathtt{p}(\mathtt{cid}, \mathtt{set})$ notification from MBRSHP, end-point `p` stores $\langle$`cid`, `set`$\rangle$ in the variable `start_change`, informs CO_RFIFO that it wishes to maintain reliable communication to the end-points in `current_view` $\cup$ `set`, and then sends a synchronization message tagged with `cid` to every end-point in `set`. The synchronization message contains `p`'s current view `v` and a `cut`, which is a mapping from processes to indices; `cut(q)` is the index of the last message from `q` that `p` *commits* to deliver before delivering any view $\mathtt{v}'$ with $\mathtt{v}'.\mathtt{startId}(\mathtt{p}) = \mathtt{cid}$. In order to preserve liveness, `p` does not commit to deliver messages that it does not already have in `msgs[q][current_view]`.

End-point `p` stores the synchronization message from `q` tagged with `cid` in `sync_msg[q][cid]`. Until `p` receives a view from MBRSHP, it does not know which synchronization messages from others to consider, so it restricts delivery of application messages to only those identified in its own latest `cut`. When a MBRSHP view $\mathtt{v}'$ is delivered to `p`, the $\mathtt{v}'.\mathtt{startId}$ mapping tells `p` to use the synchronization messages $\mathtt{sync\_msg}[\mathtt{q}][\mathtt{v}'.\mathtt{startId}(\mathtt{q})]$ from $\mathtt{q} \in \mathtt{v}'.\mathtt{set}$. The members of `p`'s transitional set for view $\mathtt{v}'$ are those end-points `q` whose $\mathtt{sync\_msg}[\mathtt{q}][\mathtt{v}'.\mathtt{startId}(\mathtt{q})].\mathtt{view}$ is the same as `p`'s current view `v`. After receiving view $\mathtt{v}'$ from MBRSHP, `p` allows delivery of application messages identified in the `cut`s conveyed in synchronization messages from processes that are already known to be members of the transitional set. The delivery of $\mathtt{view}_\mathtt{p}(\mathtt{v}', \mathtt{T})$ to `p`'s application is enabled only after `p` has received the synchronization messages from all the potential members of `T` and after it has delivered all application messages committed to by the `cut`s of the members of `T`. Since all the end-points that move from `v` to $\mathtt{v}'$ use the same set of synchronization messages, the safety properties Virtual Synchrony and Transitional Set are satisfied.

End-point `p` is guaranteed to eventually receive all the application messages sent by the members of its transitional set `T`. However, `p` may fail to receive some of the application messages sent by disconnected end-points (not in `T`) although certain cuts of members of `T` commit to deliver these messages. Such messages need to be forwarded to `p` by the members of `T` that have them. These members of `T` deduce from the `p`'s cut that `p` lacks these messages and they use a `ForwardingStrategyPredicate` to compute which of them have to forward which missing messages to `p`. We discuss possible such predictes below.

Figure 10 Virtually synchronous reliable FIFO multicast and transitional set end-point automaton.

```
AUTOMATON VS_RFIFO+TS_p  MODIFIES  WV_RFIFO_p

Signature Extension:
 Input:  send_p(m) modifies wv_rfifo.send_p(m)
         mbrshp.start_change_p(id, set), StartChangeId id, SetOf(Proc) set new
         co_rfifo.deliver_q,p(m), Proc q, SyncMsg m  new

 Output: deliver_p(q, m) modifies wv_rfifo.deliver_p(q, m)
         view_p(v, T), SetOf(Proc) T  modifies wv_rfifo.view_p(v)
         co_rfifo.reliable_p(set), SetOf(Proc) set modifies wv_rfifo.co_rfifo.reliable_p(set)
         co_rfifo.send_p(set, m), SetOf(Proc) set, SyncMsg m new
         co_rfifo.send_p(set, m) modifies wv_rfifo.co_rfifo.send_p(set, m), FwdMsg m

State Extension:
 (StartChangeId × SetOf(Proc))_⊥ start_change, initially ⊥
 For all Proc q, ViewId id:  (View v, (Proc→Int) cut)_⊥ sync_msg[q][id], initially ⊥
 SetOf(FwdMsg) forwarded_set, initially empty

Transition Restriction:
 INPUT  mbrshp.start_change_p(id, set)
 eff: start_change ← ⟨id, set⟩

 OUTPUT  co_rfifo.reliable_p(set)
 pre: start_change = ⊥ ⇒ set = current_view.set
      start_change ≠ ⊥ ⇒ set = current_view.set ∪ start_change.set

 OUTPUT  co_rfifo.send_p(set, tag=sync_msg, cid, v, cut)
 pre: start_change ≠ ⊥
      start_change.set ⊆ reliable_set
      ⟨cid, set⟩ = ⟨start_change.id, start_change.set - {p}⟩
      sync_msg[p][cid] = ⊥  ∧      v = current_view
      (∀ q ∈ current_view.set) cut(q) = LongestPrefixOf(msgs[q][v])
 eff: sync_msg[p][cid]  ← ⟨v, cut⟩

 INPUT  co_rfifo.deliver_q,p(tag=sync_msg, cid, v, cut)
 eff: sync_msg[q][cid]  ← ⟨v, cut⟩

 OUTPUT  deliver_p(q, m)
 pre:  if (start_change ≠ ⊥ ∧ sync_msg[p][start_change.id] ≠ ⊥) then
         if start_change.id ≠ mbrshp_view.startId(p) then
            last_dlvrd[q]+1 ≤ sync_msg[p][start_change.id].cut(q)
         else  let S = {r ∈ mbrshp_view.set ∩ current_view.set |
                           sync_msg[r][mbrshp_view.startId(r)].view = current_view}
              last_dlvrd[q]+1 ≤ max_{r ∈ S} sync_msg[r][mbrshp_view.startId(r)].cut(q)

 OUTPUT  view_p(v, T)
 pre: v.startId(p) = start_change.id    // to prevent delivery of obsolete views
      (∀ q ∈ v.set ∩ current_view.set) sync_msg[q][v.startId(q)] ≠ ⊥
      T = {q ∈ v.set ∩ current_view.set | sync_msg[q][v.startId(q)].view = current_view}
      (∀ q ∈ current_view.set) last_dlvrd[q] = max_{r ∈ T} sync_msg[r][v.startId(r)].cut(q)
 eff: start_change ← ⊥

 OUTPUT  co_rfifo.send_p(set,tag=fwd_msg,r,v,m,i)
 pre: (∀ q ∈ set) ⟨q, r, v, i⟩ ∉ forwarded_set
      ForwardStrategyPredicate(⟨set, r, v, i⟩, current_state)
 eff: (∀ q ∈ set) add ⟨q, r, v, m, i⟩ to forwarded_set
```

5.2.2 Forwarding Strategy Predicate

In this section we provide two examples of `ForwardingStrategyPredicate`s. With the first, multiple copies of the same message are forwarded by different end-points (Recall that one end-point does not forward the same message to a second end-point more than once. This is due to the use of `forwarded_set` in the CO_RFIFO.`send` action.) The second strategy minimizes the number of forwarded copies of a message. Obviously, many other possible strategies exist. For example, a strategy can employ randomization to decide whether an end-point should forward a message in a certain time slice, and suppress the forwarding of messages that have already been forwarded by others.

A simple strategy

According to our first example strategy, a process `p` forwards a message `m` only if `p` has committed to deliver `m`. In addition, if `m` was originally sent in view `v`, `p` forwards `m` to a process `q` only if `p` does not know of any later view of `q` than `v`, and if the latest `sync_msg` from `q` sent in view `v` indicates that `q` has not received message `m`. The strategy is defined as follows:

```
ForwardingStrategyPredicate(set, r, v, i, s[p]) =
pre: (∃ cid) s[p].sync_msg[p][cid].view = v
     i ≤ s[p].sync_msg[p][cid].cut(r)
     set = { q |  s[p].view_msg[q] ≤ v
                  (∃ cid′) s[p].sync_msg[q][cid′].view = v
                 ∧ (∄ cid′′ > cid′) s[p].sync_msg[q][cid′′].view = v
                 ∧ s[p].sync_msg[q][cid′].cut(r) < i }
```

According to this strategy, if some process `q` is missing a certain message `m`, `m` will be forwarded to `q` as soon as some other end-point that has committed to deliver `m` learns from `q`'s `sync_msg` that `q` misses it.

Since `p` has to be committed to deliver the messages that it forwards, there is only a finite number of such messages. We rely on the fact that only a finite number of messages are forwarded in order to argue that if the strategy enables forwarding of a message `m`, them `m` is eventually forwarded. Otherwise, `p` may keep receiving and forwarding later messages instead of `m`[1].

Minimizing the number of forwarded copies of a message

The second `ForwardingStrategyPredicate` we present uses the transitional set in order to agree which message should be forwarded by which member of the transitional set. Assume that `T` is known to be the transitional set for moving from view `v` to `v′`. Assume further that a member of `T` misses a message `m` that was originally sent in view `v` by a non-member of `T` and some other members of `T` have committed to deliver `m`. The strategy uses the `sync_msgs` to deterministically decide which of those members of T that have committed to deliver `m` will forward it. We use `min` function to deterministically select one such end-point. More involved functions can take into account topology of the network, communication cost, etc.

Note that although `q`'s `sync_msg` may indicate that `q` misses a message from a member of `T`, `q` will eventually receive this message from its original sender, hence there is no need to forward it. The strategy is as follows:

```
ForwardingStrategyPredicate(set, r, v, i, s[p]) =
pre: v = s[p].mbrshp_view // the current membership view can be same as current_view
```

[1] Note that we do not specify that messages are forwarded in FIFO order.

```
s[p].sync_msg[p][v.startId(p)] ≠ ⊥ // already sent own sync_msg
Let I = v.set ∩ s[p].sync_msg[p][v.startId(p)].view.set
(∀ q ∈ I) s[p].sync_msg[q][v.startId(q)] ≠ ⊥ // already rcvd right sync_msgs from potential T
Let T = {q ∈ I | s[p].sync_msg[q][v.startId(q)].view = s[p].sync_msg[p][v.startId(p)].view}
r ∉ T  // only forward messages from end-point not in T.
set = {u ∈ T  | s[p].sync_msg[u][v.startId(u)].cut(r) < i }
p = min{u ∈ T  | s[p].sync_msg[u][v.startId(u)].cut(r) ≥ i }
```

Note that in a situation when q needs to recover a message m in order to move into v' but another process p has already moved into a view $v'' > v'$, p will *not* forward m to q. However, this does not violate liveness since in this case v' is obsolete (p has already moved from it to a later view). In case v' is not obsolete, (i.e., if none of the members of v' deliver later views than v'), then this strategy guarantees that if q needs to recover a message m in order to move into v', then there will be at least one end-point p that will forward m to q reliably via a CO_RFIFO channel.

Usually, only one copy of m will be sent, but this is not always the case. In some cases when MBRSHP provides different mbrshp_views to different processes, more than one end-point may forward the same message. However, scenarios that would cause this are not dominant.

Note that with this strategy an end-point waits to receive a mbrshp_view and all the right sync_msgs before it forwards any messages. End-point q has to wait until it receives this message before it is able to proceed in to the next view.

5.2.3 Liveness of VS_RFIFO+TS

Consider a fair execution, α, in which the same MBRSHP view v' is delivered to all its members, with no subsequent MBRSHP events. In order for VS_RFIFO+TS to be live in α, the preconditions for $\mathtt{view_p}(v', *)$ should eventually be satisfied for every $p \in v'.\mathtt{set}$. There are three preconditions on view delivery:

1. The precondition $\mathtt{v.startId(p)} = \mathtt{start\_change.id}$ is satisfied from our assumption that no start_change events occur at p after p receives v'.

2. p needs to have received synchronization messages tagged with the "right" cid from all the processes in $\mathtt{v.set} \cap v'.\mathtt{set}$. This precondition is eventually satisfied since, from our assumption, every process q in $\mathtt{v.set} \cap v'.\mathtt{set}$ receives view v' from MBRSHP. Hence, a preceding $\mathtt{start\_change_q}(v'.\mathtt{startId(q)}, \mathtt{set})$ occurs, and since α is fair, q sends a synchronization message tagged with $v'.\mathtt{startId(q)}$ while p is in q's reliable set, and p receives this message.

3. The last delivered message from every q is equal to:
$\mathtt{max}_{r \in T}\mathtt{sync\_msg[r][mbrshp\_view.startId(r)].cut(q)}$. This condition is satisfied if p did not deliver any messages beyond those committed to in the cuts of the members of its transitional set T_p (as follows from the precondition on application message delivery) and if p eventually receives all the messages committed to in the cuts of the members of T_p. Note that for each such message, there is at least one end-point in T_p that has the message in its msgs buffer. If the original sender of the message, q, is in T_p, then p eventually receives the message since CO_RFIFO provides a live and reliable service from q to p (as p is in both CO_RFIFO.reliable_set[p] and CO_RFIFO.live_set[p]). Otherwise, one of the members of T_p that has the message forwards it to p (according to the ForwardingStrategyPredicate) on behalf of q, and p receives it.

5.2.4 Optimizations

Notice that end-point p does not need to send its current view and its cut to end-points which are not in current_view.set because p cannot be included in their transitional sets. Nevertheless, these end-points may wait to hear from p as p may still be in their current views. Therefore, in our algorithm, p sends synchronization messages to all the end-points in start_change.set. As an optimization, p could send a different (smaller) synchronization message to processes in start_change.set − current_view.set, containing its start_change.id only (but neither a view nor a cut). This message would be interpreted as saying "I am not in your transitional set", and the recipients would not include p in their transitional sets for views v′ with v′.startId(p) = p's start_change.id. When using this optimization, p also does not need to include its current view in synchronization messages sent to current_view.set − start_change.set, since the view can be deduced from a preceding view_msg that p sent to them.

Another optimization can be used to minimize synchronization message sizes if we strengthen the membership specification to require a MBRSHP.start_change to be sent every time the membership changes its mind about the next view. In this case, the latest MBRSHP.start_change has the same membership as the delivered MBRSHP.view, and therefore the synchronization messages do not need to include information about messages delivered from end-points in start_change.set ∩ current_view.set because the synchronization message from each of these end-points can terminate a stream of application messages that this end-point would deliver in its current view.

5.3 Adding support for Self Delivery

As a final step in constructing the automaton that models an end-point of our group communication service, GCS_p, we add support for Self Delivery to the VS_RFIFO+TS_p automaton presented above. Self Delivery requires each end-point to deliver to its application all the messages the application sends in a view, before moving on to the next view.

In order to implement Self Delivery and Virtual Synchrony together in a live manner, each end-point must *block* its application from sending new messages while a view change is taking place (as proven in [19]). Therefore, we modify VS_RFIFO+TS_p to have an output action block and an input action block_ok, and we assume that the application at end-point p has the matching actions and that it eventually responds to every block request with a block_ok response and subsequently refrains from sending messages until a view is delivered to it. In Section 6.4, we formalize this requirement by specifying an abstract client automaton.

The GCS_p automaton appears in Figure 11. After receiving the first start_change notification in a given view, end-point p issues a block request to the application and awaits receiving a block_ok response before sending a synchronization message to other members of start_change.set. The cut sent in the synchronization message commits to all the messages p received from its application in the current view.

Since the application is required to respond with block_ok and is then blocked from sending further messages, and since the p's cut commits to all the messages the application has sent in the current view, the set of messages agreed upon based on the cuts includes all of p's messages. Therefore, p delivers all these messages before moving on to a new view, and Self Delivery is satisfied. Since end-point p has its own messages on the msgs[p][p] queue, the modification does not affect the liveness property of VS_RFIFO+TS. Finally, we note that due to the use of inheritance, the GCS_p automaton satisfies all the properties we have specified in Secion 4.

Figure 11 $\texttt{GCS}_\texttt{p}$ end-point automaton: Adding support for Self Delivery.

AUTOMATON GCS_p = VS_RFIFO+TS+SD_p MODIFIES VS_RFIFO+TS_p

Signature Extension:

```
Input:  block_okp() new

Output: blockp() new
        viewp(v, T) modifies vs_rfifo+ts.viewp(v, T)
        co_rfifo.sendp(set, m) modifies vs_rfifo+ts.co_rfifo.sendp(set, m), SyncMsg m
```

State Extension:

block_status $\in$ {unblocked, requested, blocked}, initially unblocked

Transition Restriction:

```
OUTPUT  blockp()
pre: start_change ≠ ⊥
     block_status = unblocked
eff: block_status ← requested

INPUT  block_okp()
eff: block_status ← blocked

OUTPUT  viewp(v,T)
eff: block_status ← unblocked

OUTPUT  co_rfifo.sendp(set, tag=sync_msg, cid, v, cut)
pre: block_status = blocked
```

6 Correctness Proof: Safety Properties

We now prove that the safety properties of our algorithms are satisfied using invariant assertions and simulations. The safety proof is *modular*: we exploit the inheritance-based structure of our specifications and algorithms to reuse proofs. In Section 6.1 we prove correctness of the within-view reliable FIFO multicast service by showing a refinement mapping from WV_RFIFO to WV_RFIFO : SPEC. In Section 6.2 we extend this refinement mapping to map the new state added in VS_RFIFO+TS to that in VS_RFIFO : SPEC. In Section 6.3 we prove that VS_RFIFO+TS also simulates TS : SPEC. Finally, in Section 6.4 we extend the refinement above to map the new state of GCS to that of SELF : SPEC.

6.1 Within-view reliable FIFO multicast

Intuitively, in order to simulate WV_RFIFO : SPEC with WV_RFIFO, we need to show that WV_RFIFO satisfies Self Inclusion and Local Monotonicity on delivered views and we need to show that the i'th message delivered by `q` from `p` in view `v` is the i'th message sent in view `v` by the application client at `p`. Showing Self Inclusion and Local Monotonicity is straightforward. In order to prove the latter, we need to show that the algorithms correctly associate messages with the views in which they were sent, and their indices in the sequences of messages sent in these views. We prove this using invariant assertions; we then present a refinement mapping from WV_RFIFO to WV_RFIFO : SPEC and the non-trivial steps of its simulation proof.

6.1.1 Invariants

The following invariant captures the Self-Inclusion property.

Invariant 6.1 *In every reachable state* `s` *of* WV_RFIFO, *for all* `Proc p`, `p` $\in$ `s[p].mbrshp_view.set` *and* `p` $\in$ `s[p].current_view.set`.

Proof 6.1: Straightforward induction based on the MBRSHP specification. ■

In any view, before an end-point sends a `view_msg` to others (and hence before it sends any application message to others) it tells CO_RFIFO to maintain reliable connection to everybody in its current view. The following invariant captures this property.

Invariant 6.2 *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p`*, if* `s[p].current_view` $=$ `s[p].view_msg[p]`*, then* `s[p].current_view.set` $\subseteq$ `s[p].reliable_set`*.*

Proof 6.2: Straightforward induction. ■

The fact that the views delivered by an end-point satisfy Local Monotonicity is straightforward because it is a direct outcome of the underlying MBRSHP service satisfying this property. Therefore, a stream of `view_msg`s that an end-point sends to others is also monotonic. The following invariant captures this property.

Invariant 6.3 *Let* `s` *be a reachable state of* WV_RFIFO*. Consider the subsequence of messages in* `s.channel[p][q]` *for which* `m.tag=`**view_msg**. *We examine the sequence of views included in these view messages, and construct a new sequence* seq *of views by pre-pending this view sequence with the element* `s[q].view_msg[p]`*. For all* `Proc p, Proc q`*, the following propositions are true:*

1. *The sequence* seq *is (strictly) monotonically increasing.*
2. *If* `s[p].current_view` $\neq$ `s[p].view_msg[p]`*, then* `s[p].current_view` *is strictly greater then the last (largest) element of* seq*.*
3. *If* `s[p].current_view` $=$ `s[p].view_msg[p]`*, and if* `q` $\in$ `s[p].current_view.set`*, then* `s[p].current_view` *is equal to the last (largest) element of* seq*.*

Proof 6.3: All three propositions are true in the initial state. We now consider steps involving the critical actions:

CO_RFIFO.$\mathtt{lose(p,q)}$: The first two propositions remain true because this action throws away only the last message from the CO_RFIFO `s.channel[p][q]`.

The third proposition is vacuously true because `q` can not be in `s[p].current_view.set`. If it were, the CO_RFIFO.$\mathtt{lose(p,q)}$ action would not be enabled because Invariant 6.2 would imply that `s[p].current_view.set` is a subset of `s[p].reliable_set`, which would then imply that `q` $\in$ `s.reliable_set[p]` (because `s[p].reliable_set` $=$ `s.reliable_set[p]`, as can be shown by straightforward induction).

$\mathtt{view_p(v)}$: The first proposition is unaffected. The second proposition follows from the inductive hypothesis and the precondition `v.id` $>$ `s[p].current_view.id`. The third proposition is vacuously true because `s[p].current_view` $\neq$ `s[p].view_msg[p]` as follows from the precondition `v.id` $>$ `s[p].current_view.id` and the fact that, in every reachable state `s`, `s[p].current_view` $\geq$ `s[p].view_msg[p]` (can be proven by straightforward induction).

CO_RFIFO.$\mathtt{send_p(set, tag} = \textbf{view\_msg}\mathtt{, v)}$: The first proposition is true in the post-state because of the inductive hypothesis of the second proposition. The second proposition is vacuously true in the post-state. The third proposition is true in the post-state because of the effect of this action.

CO_RFIFO.$\mathtt{deliver_{p,q}(tag} = \textbf{view\_msg}\mathtt{, v)}$: It is straightforward to see that all three propositions remain true in the post-state.

■

In order to reason about original application messages traveling on CO_RFIFO channels we need a way of referencing the views in which they were sent and their FIFO indices. To this end we augment each original application message ⟨ `tag=`**app_msg**`, m` ⟩ with two *history tags*, `Hv` and `Hi`, that are set to `current_view` and `last_sent + 1` resp. when CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \mathtt{tag} = \mathbf{app\_msg}, \mathtt{m})$ occurs. (See Appendix A for details on history variables).

```
OUTPUT co_rfifo.send_p(set, tag=app_msg, m, Hv, Hi)
pre: ...
     Hv = current_view
     Hi = last_sent + 1
eff: ...
```

With the addition of these history tags, the interface between WV_RFIFO and CO_RFIFO for handling original application messages becomes CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \mathtt{tag} = \mathbf{app\_msg}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi})$ and CO_RFIFO.$\mathtt{deliver_{p,q}}(\mathtt{tag} = \mathbf{app\_msg}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi})$.

We show that, when process `q` receives an application message `m` tagged with a history view `Hv` and a history index `Hi`, the current value of `q`'s `view_msg[p]` equals `Hv` and that of `last_rcvd[p] + 1` equals `Hi`. The following two invariants capture this relationship:

Invariant 6.4 *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p, Proc q`*, the following is true:*

For all messages ⟨ `tag=`**app_msg**`, m, Hv, Hi` ⟩ *on the* CO_RFIFO `s.channel[p][q]`*, view* `Hv` *equals either the view of the closest preceding view message on* `s.channel[p][q]` *if there is such, or* `s[q].view_msg[p]` *otherwise.*

Proof 6.4: By induction. The step involving a CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \mathtt{tag} = \mathbf{app\_msg}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi})$ action follows directly from Invariant 6.3 Part 3. The proposition is not affected by steps involving CO_RFIFO.`lose(p,q)` because those may only remove the last messages from the CO_RFIFO `s.channel[p][q]`. The other steps are straightforward. ■

Invariant 6.5 *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p` *and* `Proc q`*, the following is true:*

The history index attached to an original application message `m` *sent in a view* `Hv` *that is in transit on a* CO_RFIFO *channel to end-point* `q` *is equal to the number of such messages (including* `m`*) that precede* `m` *on that channel, plus those (if any) that* `q` *has already received.*

Formally, if ⟨ `tag=`**app_msg**`, m, Hv, Hi` ⟩ = `s.channel[p][q][j]` *for some index* `j`*, then*

$$
\begin{aligned}
&Hi = \\
&\quad \left|\{\mathtt{msg} \in \mathtt{s.channel[p][q][\ ..\ j]}\ :\ \mathtt{msg.tag{=}}\mathbf{app\_msg}\ \mathtt{and}\ \mathtt{msg.Hv = Hv}\}\right| + \\
&\quad + \begin{cases} \mathtt{s[q].last\_rcvd[p]} & \textit{if}\ \mathtt{s[q].view\_msg[p] = Hv} \\ 0 & \textit{otherwise} \end{cases}
\end{aligned}
$$

Proof 6.5: In the initial state the proposition is vacuously true because `s.channel[p][q]` is empty.

<u>CO_RFIFO.`lose(p,q)`:</u> The proposition remains true since CO_RFIFO.`lose(p,q)` discards only the last messages from the CO_RFIFO `s.channel[p][q]`.

$\underline{\text{CO\_RFIFO.}\mathtt{deliver}_{p,q}(\mathtt{tag} = \mathbf{view\_msg}, \mathtt{v})}$: We have to consider the effects on two types of application messages: those associated with view `s[q].view_msg[p]`, and those associated with view `Hv`. Invariants 6.3 Part 1 and 6.4 imply that there are no application messages with `msg.Hv` = `s[q].view_msg[p]` on the CO_RFIFO `channel[p][q]`. Thus, the proposition does not apply for such messages. For those messages that have `msg.Hv = Hv`, the proposition remains true because $s'[q].\mathtt{last\_rcvd}[p]$ is set to 0 as a result of this action.

$\underline{\text{CO\_RFIFO.}\mathtt{deliver}_{p,q}(\mathtt{tag} = \mathbf{app\_msg}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi})}$: Follows immediately from the effect of this action, the inductive hypothesis, and Invariant 6.4.

$\underline{\text{CO\_RFIFO.}\mathtt{send}_{p}(\mathtt{set}, \mathtt{tag} = \mathbf{app\_msg}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi})}$: The inductive step follows immediately from the inductive hypothesis if we can prove the following proposition for $\mathtt{q} \in \mathtt{s[p].current\_view.set}$:

$$
\begin{aligned}
&\mathtt{s[p].last\_sent} = \\
&\quad \left| \{ \mathtt{msg} \in \mathtt{s.channel[p][q]} \ : \ \mathtt{msg.tag}{=}\mathbf{app\_msg} \ \mathtt{and} \ \mathtt{msg.Hv} = \mathtt{s[p].current\_view} \} \right| + \\
&\quad + \begin{cases} \mathtt{s[q].last\_rcvd[p]} & \text{if } \mathtt{s[q].view\_msg[p]} = \mathtt{s[p].current\_view} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

i.e., that the value of `s[p].last_sent` equals to the number of application messages that `p` sent in its current view and that are either still in transit on the CO_RFIFO `s.channel[p][q]` or already received by `q`.

This proposition is proven by induction as follows.

- Assume that the last message on `s.channel[p][q]` is an application message `msg` with `msg.Hv` = `s[p].current_view`. If a step involving CO_RFIFO.lose(p,q) action could occur, then the proposition would be false. However, as we are going to argue now, $\mathtt{q} \in \mathtt{s.reliable\_set[p]}$, so such a step cannot occur.
  We can prove by straightforward induction that the fact that `msg` is in `s.channel[p][q]` implies that `s[p].view_msg[p]` = `s[p].current_view`. By invariant 6.2 , $\mathtt{s[p].current\_view.set} \subseteq \mathtt{s[p].reliable\_set}$. Since $\mathtt{q} \in \mathtt{s[p].current\_view.set}$, then $\mathtt{q} \in \mathtt{s.reliable\_set[p]}$ (because `s[p].reliable_set` = `s.reliable_set[p]`).
- The proposition remains true for steps involving $\mathtt{view}_p(\mathtt{v})$ action because its effect sets $s'[p].\mathtt{last\_sent}$ to 0 and because both summands of the right hand side of the equation also becomes 0. Indeed, the first summand becomes 0 because CO_RFIFO channels never have messages tagged with views that are larger then the current views of the messages' senders (as can be shown by a simple inductive proof); the second summand becomes 0 because Invariant 6.3 Part 2 implies that $s'[q].\mathtt{view\_msg}[p] \neq s'[p].\mathtt{current\_view}$.
- The proposition remains true for steps involving CO_RFIFO.$\mathtt{deliver}_{p,q}(\mathtt{tag} = \mathbf{view\_msg}, \mathtt{v})$ action because $\mathtt{s[q].view\_msg[p]} \neq \mathtt{s[p].current\_view}$, as follows immediately from Invariant 6.3.
- The proposition remains true for steps involving CO_RFIFO.$\mathtt{send}_p(\mathtt{set}, \mathtt{tag} = \mathbf{app\_msg}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi})$ and CO_RFIFO.$\mathtt{deliver}_{p,q}(\mathtt{tag} = \mathbf{app\_msg}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi})$ actions follow immediately from their effects, the inductive hypotheses, and Invariant 6.4.

■

Now we can prove the following key invariant which relates application messages either in transit on the CO_RFIFO channels or at end-points' queues to the corresponding messages on the senders' queues.

Invariant 6.6 *In every reachable state* `s` *of* WV_RFIFO, *for all* `Proc p` *and* `Proc q`, *the following are true:*

1. *If* ⟨ `tag=`**app_msg**`, m, Hv, Hi` ⟩ ∈ `s.channel[p][q]`, *then* `s[p].msgs[p][Hv][Hi] = m`.
2. *If* ⟨ `tag=`**fwd_msg**`, r, m, v, i` ⟩ ∈ `s.channel[p][q]`, *then* `s[r].msgs[r][v][i] = m`.
3. *If* `s[q].msgs[p][v][i] = m`, *then* `s[p].msgs[p][v][i] = m`.

Proof 6.6:

Basis: The propositions are vacuously true in the initial state because all message queues are empty.

Inductive Step: The following are the critical actions:

$\texttt{send}_{\texttt{p}}\texttt{(m)}$,
$\texttt{co\_rfifo.send}_{\texttt{p}}\texttt{(set, tag=}\textbf{app\_msg}\texttt{, m, Hv, Hi)}$,
$\texttt{co\_rfifo.deliver}_{\texttt{q,p}}\texttt{(tag=}\textbf{app\_msg}\texttt{, m, Hv, Hi)}$,
$\texttt{co\_rfifo.send}_{\texttt{p}}\texttt{(set, tag=}\textbf{fwd\_msg}\texttt{, r, v, m, i)}$,
$\texttt{co\_rfifo.deliver}_{\texttt{q,p}}\texttt{(tag=}\textbf{fwd\_msg}\texttt{, r, v, m, i)}$.

Inductive steps involving each of these critical actions are straightforward. For steps involving $\text{CO\_RFIFO}.\texttt{deliver}_{\texttt{q,p}}(\texttt{tag} = \textbf{app\_msg}, \texttt{m}, \texttt{Hv}, \texttt{Hi})$, we have to use Invariants 6.4 and Invariant 6.5, which respectively imply that history view `Hv` equals `s[p].view_msg[q]` and that history index `Hi` equals `s[p].last_rcvd[q] + 1`. ■

6.1.2 Simulation

Lemma 6.1 *The following function* `R()` *is a* refinement mapping *from automaton* WV_RFIFO *to automaton* WV_RFIFO : SPEC *with respect to their reachable states.*

`R(s:` WV_RFIFO`)` → WV_RFIFO : SPEC =

`For each Proc p, View v:`	`msgs[p][v]`	=	`s[p].msgs[p][v]`
`For each Proc p, Proc q:`	`last_dlvrd[p][q]`	=	`s[q].last_dlvrd[p]`
`For each Proc p:`	`current_view[p]`	=	`s[p].current_view`

Proof 6.1:

Action Correspondence: Automaton WV_RFIFO : SPEC has three types of actions. Actions of the types $\texttt{view}_{\texttt{p}}(\texttt{v})$, $\texttt{send}_{\texttt{p}}(\texttt{m})$, and $\texttt{deliver}_{\texttt{p}}(\texttt{q},\texttt{m})$, are simulated when WV_RFIFO takes respectively the corresponding actions $\texttt{view}_{\texttt{p}}(\texttt{v})$, $\texttt{send}_{\texttt{p}}(\texttt{m})$, and $\texttt{deliver}_{\texttt{p}}(\texttt{q},\texttt{m})$. Steps of WV_RFIFO involving other actions correspond to empty steps of WV_RFIFO : SPEC.

Simulation Proof: In the most part the simulation proof is straightforward. Here, we present only the interesting steps:

The fact that the corresponding step of WV_RFIFO : SPEC is enabled when WV_RFIFO takes a step involving $\texttt{view}_{\texttt{p}}(\texttt{v})$ relies on $\texttt{p} \in \texttt{mbrshp\_view.set}$ (Invariant 6.1).

For steps involving $\texttt{deliver}_{\texttt{p}}(\texttt{q},\texttt{m})$, to deduce that the corresponding step of WV_RFIFO : SPEC is enabled, we need to know that the message at index `s[p].last_dlvrd[q] + 1` at end-point `p`'s `s[p].msgs[q][s[p].current_view]` is the same message that end-point `q` has on its corresponding queue at the same index. This property is implied by Invariant 6.6 Part 3.

Steps that involve receiving original and forwarded application messages from the network simulate empty steps of WV_RFIFO : SPEC. Among these steps the only critical ones are those that deliver a message from p to p because they may affect s[p].msgs[p][p] queue. Since end-points do not send original application messages to themselves, such steps involving original messages may not happen; This can be proven by straightforward induction, as the only critical action CO_RFIFO.$\mathtt{send}_\mathtt{p}$(set, tag = **app_msg**, m) is preconditioned by set = s[p].current_view.set - {p}. Unlike original messages, forwarded messages may theoretically be sent and delivered to their original end-points. However, as Invariant 6.6 Part 2 shows, these messages are the same as those they replace, thus the message queue is not affected. ■

Lemma 6.1 establishes function R() as a refinement mapping from the algorithm automaton WV_RFIFO to the specification automaton WV_RFIFO : SPEC. It allows us to conclude the following Theorem.

Theorem 6.1 *Automaton* WV_RFIFO *implements automaton* WV_RFIFO : SPEC *in the sense of trace inclusion.*

Proof 6.1: Follows immediately from Lemma 6.1. ■

6.2 Virtual Synchrony

We now show that automaton VS_RFIFO+TS simulates VS_RFIFO : SPEC. We prove this by extending the refinement above using the Proof Extension Theorem of [26] (see Appendix A for details).

6.2.1 Invariants

We need to prove that end-points that move together from one view to the next use the same sets of cuts (i.e., compute the same transitional sets and use the same synchronization messages from the members of the transitional set).

Invariant 6.7 *In every reachable state* s *of* VS_RFIFO+TS, *for all* Proc p, Proc q, *and for every* StartChangeId cid, *if* e[q].sync_msg[p][cid] $\neq \perp$, *then:*
e[q].sync_msg[p][cid] = s[p].sync_msg[p][cid].

Proof 6.7: The proposition is true in the initial state s_0 as all s_0[q].sync_msg[p][cid] = $\perp$. The inductive step involving a CO_RFIFO.$\mathtt{send}_\mathtt{p}$(set, tag = **sync_msg**, cid, v, cut) action is trivial. The inductive step involving a CO_RFIFO.$\mathtt{deliver}_{\mathtt{p,q}}$(tag = **sync_msg**, cid, v, cut) action follows immediately from the following proposition:

$$\langle \mathtt{tag{=}\mathbf{sync\_msg},\ cid,\ v,\ cut} \rangle \in \mathtt{s.channel[p][q]} \Rightarrow \mathtt{s[p].sync\_msg[p][cid]} = \langle \mathtt{v,\ cut} \rangle,$$

which can be proven by straightforward induction. ■

Corollary 6.1 *End-points that move together from one view to the next, use the same sets of cuts.*

Proof : Consider two end-points that at different times install a view $\mathtt{v}'$ while in a view $\mathtt{v}$. At the time of installing view $\mathtt{v}'$, each of these end-points has synchronization messages from all end-points in the intersection of these views (first precondition), and these synchronization messages are the same as those at their original end-points (Invariant 6.7). Thus, the two end-points calculate the same transitional sets (see how transitional set is calculated), and use the same cuts from the members of this transitional set. ■

6.2.2 Simulation

We augment VS_RFIFO+TS with a *global* history variable H_cut that keeps track of the cuts used for moving between views.

```
For each View v, v': (Proc → Int)⊥ H_cut[v][v'], initially ⊥

OUTPUT view_p(v, T) modifies wv_rfifo.view_p(v)
pre: ...
eff: ...
     H_cut[current_view][v] ← max_{r ∈ T} sync_msg[r][v.startId(r)].cut[q]
```

We now extend the refinement mapping R() of Lemma 6.1 with the following new mapping $\mathtt{R_n}()$:

$$\texttt{For each View v, v}': \mathtt{cut}[\mathtt{v}][\mathtt{v}'] \;=\; \mathtt{s.H\_cut}[\mathtt{v}][\mathtt{v}'].$$

We call the resulting mapping $\mathtt{R}'()$. We exploit the Proof Extension Theorem of [26] (see Appendix A for details) in order to prove that $\mathtt{R}'()$ is a refinement mapping from automaton VS_RFIFO+TS to automaton VS_RFIFO : SPEC.

Lemma 6.2 *The mapping* $\mathtt{R}'()$ *defined above is a refinement mapping from automaton* VS_RFIFO+TS *to automaton* VS_RFIFO : SPEC.

Proof 6.2:

Action Correspondence: The action correspondence of WV_RFIFO is modified as follows: Consider steps $(\mathtt{s}, \mathtt{view_p}(\mathtt{v}', \mathtt{T}), \mathtt{s}')$ of VS_RFIFO+TS which involve delivering views to the application clients. Among these steps, those that are the first to set variable $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$ (when $\mathtt{s.H\_cut}[\mathtt{v}][\mathtt{v}'] = \perp$) simulate two steps of VS_RFIFO : SPEC, a step involving $\mathtt{set\_cut}(\mathtt{v}, \mathtt{v}', \mathtt{s}'.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}'])$, followed by a step involving $\mathtt{view_p}(\mathtt{v}')$. The rest (when $\mathtt{s.H\_cut}[\mathtt{v}][\mathtt{v}'] \neq \perp$) simulate single steps that involve just $\mathtt{view_p}(\mathtt{v}')$.

Simulation Proof:

First, we have to show that the refinement mapping of WV_RFIFO (presented in Lemma 6.1) is still preserved after the modifications introduced by VS_RFIFO : SPEC to WV_RFIFO : SPEC. Automaton VS_RFIFO : SPEC adds the following preconditions to $\mathtt{view_p}(\mathtt{v}')$ actions of WV_RFIFO : SPEC:

```
cut[current_view[p]][v] ≠ ⊥
(∀ q) last_dlvrd[q][p] = cut[current_view[p]][v](q)
```

Using the VS_RFIFO+TS automaton code, the extended mapping $\mathtt{R}'()$, and the action correspondence above, it is straightforward to show that both of these preconditions are satisfied when a step of VS_RFIFO+TS involving $\mathtt{view_p}(\mathtt{v}', \mathtt{T})$ attempts to simulate a step of VS_RFIFO : SPEC involving $\mathtt{view_p}(\mathtt{v}')$.

Second, we have to show that the mapping $\mathtt{R_n}()$ used to extend $\mathtt{R()}$ to $\mathtt{R}'()$ is also a refinement. For those steps $(\mathtt{s}, \mathtt{view_p}(\mathtt{v}', \mathtt{T}), \mathtt{s}')$ that are the first to set variable $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$, the action correspondence implies that the mapping is preserved. For those steps that are not the first to set variable $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$, the mapping is preserved because $\mathtt{s}'.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}'] = \mathtt{s}.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$, as follows from Corollary 6.1. ■

Lemma 6.2 establishes function $\mathtt{R}'()$ as a refinement mapping from the algorithm automaton VS_RFIFO+TS to the specification automaton VS_RFIFO : SPEC. It allows us to deduce the following Theorem:

Theorem 6.2 *Automaton* VS_RFIFO+TS *implements automaton* VS_RFIFO : SPEC *in the sense of trace inclusion.*

Proof 6.2: Follows immediately from Lemma 6.2. ■

6.3 Transitional Set

We now show that VS_RFIFO+TS simulates TS : SPEC. The proofs makes use of *prophecy variables.* A simulation proof that uses prophecy variables implies only finite trace inclusion (and not infinite trace inclusion). Finite trace inclusion is sufficient for proving safety properties, (see Appendix A for details).

6.3.1 Invariants

Invariant 6.8 *In every reachable state* `s` *of* VS_RFIFO+TS, *for all* `Proc p` *and* `StartChangeId id`, *if* $\mathtt{id} > \mathtt{s[MBRSHP].start\_change[p].id}$, *then* $\mathtt{s[p].sync\_msg[p][id]} = \perp$.

Proof 6.8: The proposition is true in the initial state. It remains true for the inductive step involving MBRSHP.$\mathtt{start\_change_p}(\mathtt{id}, \mathtt{set})$ because `s[mbrshp].start_change[p].id` is increased as a result of this action. For the step involving CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})$, the proposition remains true because $\mathtt{cid} = \mathtt{s[MBRSHP].start\_change[p].id}$, as implied by the precondition $\mathtt{cid} = \mathtt{s[p].start\_change.id}$ and the following invariant which can be proven by straightforward induction:

In every reachable state `s` of VS_RFIFO+TS, for all `Proc p`, if $\mathtt{s[p].start\_change.id} \neq \perp$, then $\mathtt{s[MBRSHP].start\_change[p].id} = \mathtt{s[p].start\_change.id}$. This invariant holds in the initial state. Critical action MBRSHP.$\mathtt{start\_change_p}(\mathtt{id}, \mathtt{set})$ makes it true; Critical action $\mathtt{view_p}(\mathtt{v}, \mathtt{T})$ makes it vacuously true.

Finally, a step involving CO_RFIFO.$\mathtt{deliver_{q,p}}(\mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})$ does not affect the proposition because the case `q=p` can not happen since, as can be proven by straightforward induction, end-points do not send synchronization messages to themselves. ■

Lemma 6.3 *For any step* $(\mathtt{s}, \text{MBRSHP}.\mathtt{start\_change_p}(\mathtt{id}, \mathtt{set}), \mathtt{s}')$ *of* VS_RFIFO+TS,

$$\mathtt{s[p].sync\_msg[p][start\_change.id]} = \bot.$$

Proof 6.3: Follows immediately from the precondition $\mathtt{id} > \mathtt{s[}\text{MBRSHP}\mathtt{].start\_change[p].id}$ and Invariant 6.8. ■

Invariant 6.9 *In every reachable state* s *of* VS_RFIFO+TS, *for all* Proc p, *if* s[p].start_change $\neq \bot$ *and* s[p].sync_msg[p][s[p].start_change.id] $\neq \bot$, *then*

$$\mathtt{s[p].sync\_msg[p][s[p].start\_change.id].view} = \mathtt{s[p].current\_view}.$$

Proof 6.9: The proposition is vacuously true in the initial state. For the inductive step, consider the following critical actions:

$\underline{\text{MBRSHP}.\mathtt{start\_change_p}(\mathtt{id}, \mathtt{set})}$: The proposition remains vacuously true because $\mathtt{s'[p].sync\_msg[p][start\_change.id]} = \bot$ (Lemma 6.3).

$\underline{\text{CO\_RFIFO}.\mathtt{send_p}(\mathtt{set}, \mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})}$: Follows immediately from the code.

$\underline{\text{CO\_RFIFO}.\mathtt{deliver_{q,p}}(\mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})}$: The proposition is unaffected because the case q=p can not happen since, as can be proven by straightforward induction, end-points do not send synchronization messages to themselves.

$\underline{\mathtt{view_p}(\mathtt{v})}$: The proposition becomes vacuously true because $\mathtt{s'[p].start\_change} = \bot$.

■

6.3.2 Simulation

We augment VS_RFIFO+TS with a prophecy variable P_legal_views(p)(id) for each Proc p, and each StartChangeId id. This variable is set, at the time a start_change id is delivered to an end-point p, to a *predicted* finite set of future views that are allowed to contain id as p's start_change id.

```
Prophecy Variable:
For each Proc p, StartChangeId id: SetOf(View) P_legal_views(p)(id), initially arbitrary

INTERNAL mbrshp.start_change_p(id, set)   hidden param V, a finite set of views
pre: ...
     choose V such that ∀ v ∈ V: (p ∈ v.set) ∧ (v.startId(p) = id)
eff: ...
     P_legal_views(p)(id) ← V

OUTPUT view_p(v, T)
pre: ...
     (∀ q ∈ v.set) v ∈ P_legal_views(q)(v.startId(q))
eff: ...
```

The VS_RFIFO+TS automaton augmented with the prophecy variable has the same traces as those of the original automaton because, it is straightforward to seen that the following conditions required for adding a prophecy variable hold:

1. Every state has at least one value for `P_legal_views(p)(id)`.
2. No step is disabled in the *backward direction* by new preconditions involving `P_legal_views`.
3. Values assigned to state variables do not depend on the values of `P_legal_views`.
4. If s_0 is an initial state of VS_RFIFO+TS, and $\langle s_0, \mathtt{P\_legal\_views}\rangle$ is a state of the automaton VS_RFIFO+TS augmented with the prophecy variable, then this state is an initial state.

Invariant 6.10 *In every reachable state* `s` *of* VS_RFIFO+TS*, for all* `Proc p`*, if* `s[p].start_change` $\neq \perp$*, then, for all* `View v` $\in$ `P_legal_views(p)(s[p].start_change.id)`*, it follows that* `p` $\in$ `v.set` *and* `v.startId(p)` $=$ `s[p].start_change.id`*.*

Proof 6.10: By induction. The only critical actions are MBRSHP.$\mathtt{start\_change}_{\mathtt{p}}(\mathtt{id}, \mathtt{set})$ and $\mathtt{view}_{\mathtt{p}}(\mathtt{v}, \mathtt{T})$. The proposition is true after the former, and is vacuously true after the latter. ■

Lemma 6.4 *The following function* `TS()` *is a* refinement mapping *from automaton* VS_RFIFO+TS *to automaton* TS : SPEC *with respect to their reachable states.*

`TS(s:` VS_RFIFO+TS`)` $\rightarrow$ TS : SPEC $=$

`For each Proc p: current_view[p]` $=$ `s[p].current_view`

`For each Proc p, View v: prev_view[p][v]` $=$

$$= \begin{cases} \perp & \textit{if } \mathtt{v} \notin \mathtt{s.P\_legal\_views[p][v.startId(p)]} \\ \mathtt{s[p].sync\_msg[p][v.startId(p)].view} & \textit{otherwise} \end{cases}$$

Proof 6.4:

Action Correspondence: A step (s, CO_RFIFO.$\mathtt{send}_{\mathtt{p}}(\mathtt{set}, \mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})$, s′) of VS_RFIFO+TS simulates a sequence of steps of TS : SPEC that involve one $\mathtt{set\_prev\_view}_{\mathtt{p}}(\mathtt{v}')$ for each $\mathtt{v}' \in$ `s.P_legal_views(p)(cid)`. A step (s, $\mathtt{view}_{\mathtt{p}}(\mathtt{v}, \mathtt{T})$, s′) of VS_RFIFO+TS simulates (TS(s), $\mathtt{view}_{\mathtt{p}}(\mathtt{v}, \mathtt{T})$, TS(s′)) of TS : SPEC.

Simulation Proof: Consider the following critical actions:

<u>MBRSHP.$\mathtt{start\_change}_{\mathtt{p}}(\mathtt{id}, \mathtt{set})$:</u> A step involving this action simulates an empty step of TS : SPEC. The simulation holds because `s[p].sync_msg[p][id]` $= \perp$ (Lemma 6.3).

<u>CO_RFIFO.$\mathtt{send}_{\mathtt{p}}(\mathtt{set}, \mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})$:</u> simulates a sequence of steps of TS : SPEC that involve one $\mathtt{set\_prev\_view}_{\mathtt{p}}(\mathtt{v}')$ for each $\mathtt{v}' \in$ `s.P_legal_views(p)(cid)`. Each such step is enabled as can be seen from the following derivation:

$$\begin{aligned} \mathtt{TS(s).prev\_view[p][v]} &= \mathtt{s[p].sync\_msg[p][v.startId(p)].view} \text{ (Refinement mapping)} \\ &= \mathtt{s[p].sync\_msg[p][cid].view} \text{ (Invariant 6.10)} \\ &= \perp. \text{ (Precondition)} \end{aligned}$$

The simulation step follows from the code, refinement mapping `TD()`, and the way action correspondence is defined.

<u>CO_RFIFO.$\mathtt{deliver}_{\mathtt{q,p}}(\mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})$:</u> A step involving this action does not affect any of the variables of the refinement mapping and thus simulates an empty step of TS : SPEC. In particular, note that the case of `q=p` may not happen because end-points do not send cuts to themselves; This can be shown by straightforward induction.

$\underline{\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})}$: A step involving this action simulates a step of TS : SPEC that involves $\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$. The key thing is to show that it is enabled (since it is straightforward to see that, if it is, the refinement is preserved). Action $\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$ of TS : SPEC has three preconditions. The fact that they are enabled follows directly from the inductive hypothesis, the code, the refinement mapping, and Invariants 6.9 and 6.10.

■

Lemma 6.4 establishes function `TS()` as a refinement mapping from the algorithm automaton VS_RFIFO+TS to the specification automaton TS : SPEC. It allows us to conclude the following Theorem.

Theorem 6.3 *Automaton* VS_RFIFO+TS *implements automaton* TS : SPEC *in the sense of* finite *trace inclusion.*

Proof 6.3: Follows immediately from Lemma 6.4. ■

6.4 Self Delivery

We now prove that the complete GCS end-point automaton simulates SELF : SPEC. In order to prove this, we need to formalize our assumptions about the behavior of the clients of a GCS end-point: we assume that a client eventually responds to every `block` request with a `block_ok` response and subsequently refrains from sending messages until a `view` is delivered to it. We formalize this requirement by specifying an abstract client automaton in Figure 12. In this automaton, each locally controlled action is defined to be a task by itself, which means that it eventually happens if it becomes enabled unless it is subsequently disabled by another action.

Figure 12 Abstract specification of a blocking client at end-point p

```
AUTOMATON CLIENT_p : SPEC

Signature:
 Input:  deliver_p(q, m), Proc q, AppMsg m        Output: send_p(m), AppMsg m
         view_p(v), View v                                block_ok_p()
         block_p()

State:
 block_status ∈ {unblocked, requested, blocked}, initially unblocked

Transitions:
 INPUT  block_p()                                 OUTPUT  send_p(m)
 eff: block_status ← requested                    pre: block_status ≠ blocked
                                                  eff: none
 OUTPUT  block_ok_p()
 pre: block_status = requested                    INPUT  deliver_p(q, m)
 eff: block_status ← blocked                      eff: none

                                                  INPUT  view_p(v)
                                                  eff: block_status ← unblocked
```

6.4.1 Invariants

The following invariant states that GCS end-points and their clients have the same perception of what their `block_status` is.

Invariant 6.11 *In every reachable state* $\mathtt{s}$ *of* GCS*, for all* $\mathtt{Proc\ p}$*,*

$$\mathtt{s}[\mathrm{GCS}_{\mathtt{p}}].\mathtt{block\_status} = \mathtt{s}[\mathtt{client}_{\mathtt{p}}].\mathtt{block\_status}.$$

Proof 6.11: Trivial induction. ∎

Invariant 6.12 *In every reachable state* $\mathtt{s}$ *of* GCS*, for all* $\mathtt{Proc\ p}$*, if* $\mathtt{s[p].start\_change} \neq \perp$ *and* $\mathtt{s[p].block\_status} \neq \mathtt{blocked}$*, then* $\mathtt{s[p].sync\_msg[p][s[p].start\_change.id]} = \perp$*.*

Proof 6.12: The proposition is vacuously true in the initial state $\mathtt{s}_0$ because $\mathtt{s}_0[\mathtt{p}].\mathtt{start\_change} = \perp$. For the inductive step, consider the following critical actions:

$\underline{\mathrm{MBRSHP}.\mathtt{start\_change}_{\mathtt{p}}(\mathtt{id}, \mathtt{set})}$: The proposition remains true because of Lemma 6.3.

$\underline{\mathtt{block}_{\mathtt{p}}()}$: The proposition is true in the post-state if it is true in the pre-state.

$\underline{\mathtt{block\_ok}_{\mathtt{p}}()}$: The proposition becomes vacuously true because $\mathtt{s}'[\mathtt{p}].\mathtt{block\_status} = \mathtt{blocked}$.

$\underline{\mathrm{CO\_RFIFO}.\mathtt{send}_{\mathtt{p}}(\mathtt{set}, \mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})}$: The proposition remains vacuously true because $\mathtt{s[p].block\_status} = \mathtt{s}'[\mathtt{p}].\mathtt{block\_status} = \mathtt{blocked}$.

$\underline{\mathrm{CO\_RFIFO}.\mathtt{deliver}_{\mathtt{q},\mathtt{p}}(\mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})}$: The proposition is unaffected because the case `q=p` can not happen since, as can be proven by straightforward induction, end-points do not send cuts to themselves.

$\underline{\mathtt{view}_{\mathtt{p}}(\mathtt{v}, \mathtt{T})}$: The proposition becomes vacuously true because $\mathtt{s}'[\mathtt{p}].\mathtt{start\_change} = \perp$.

∎

Invariant 6.13 *In every reachable state* $\mathtt{s}$ *of* GCS*, for all* $\mathtt{Proc\ p}$*, if* $\mathtt{s[p].start\_change} \neq \perp$ *and* $\mathtt{s[p].sync\_msg[p][s[p].start\_change.id]} \neq \perp$*,*
$\mathtt{s[p].sync\_msg[p][s[p].start\_change.id].cut[p]} =$
$\mathtt{LastIndexOf(s[p].msgs[p][s[p].current\_view])}$*.*

Proof 6.13: The proposition is vacuously true in the initial state $\mathtt{s}_0$ because $\mathtt{s}_0[\mathtt{p}].\mathtt{start\_change} = \perp$. For the inductive step, consider the following critical actions:

$\underline{\mathtt{send}_{\mathtt{p}}(\mathtt{m})}$: The proposition is vacuously true because $\mathtt{s}'[\mathtt{p}].\mathtt{sync\_msg}[\mathtt{p}][\mathtt{s}[\mathtt{p}].\mathtt{start\_change.id}] = \perp$, as follows from the precondition $\mathtt{s}[\mathtt{client}_{\mathtt{p}}].\mathtt{block\_status} \neq \mathtt{blocked}$ on this action at $\mathtt{client}_{\mathtt{p}}$, and from Invariants 6.11 and 6.12.

$\underline{\mathrm{MBRSHP}.\mathtt{start\_change}_{\mathtt{p}}(\mathtt{id}, \mathtt{set})}$: The proposition is vacuously true because $\mathtt{s}'[\mathtt{p}].\mathtt{sync\_msg}[\mathtt{p}][\mathtt{id}] = \perp$ (Lemma 6.3).

$\underline{\mathrm{CO\_RFIFO}.\mathtt{send}_{\mathtt{p}}(\mathtt{set}, \mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})}$: Follows directly from the code. More specifically from the precondition: $(\forall \mathtt{q} \in \mathtt{current\_view.set})\ \mathtt{cut(q)} = \mathtt{LongestPrefixOf(msgs[q][v])}$, and from the fact that $\mathtt{p} \in \mathtt{current\_view.set}$ (Invariant 6.1).

$\underline{\text{CO\_RFIFO}.\mathtt{deliver}_{q,p}(\mathtt{tag} = \mathbf{sync\_msg}, \mathtt{cid}, \mathtt{v}, \mathtt{cut})}$: The proposition is unaffected because the case $\mathtt{q{=}p}$ can not happen since, as can be proven by straightforward induction, end-points do not send cuts to themselves.

$\underline{\mathtt{view}_p(\mathtt{v}, \mathtt{T})}$: The proposition becomes vacuously true because $\mathtt{s}'[p].\mathtt{start\_change} = \perp$.

■

6.4.2 Simulation

Lemma 6.2 in Section 6.2 on page 27 establishes function $\mathtt{R}'()$ as a refinement mapping from automaton VS_RFIFO+TS to automaton VS_RFIFO : SPEC. We now argue that $\mathtt{R}'()$ is also a refinement mapping from automaton GCS to automaton SELF : SPEC.

Lemma 6.5 *Refinement mapping* $\mathtt{R}'()$ *from automaton* VS_RFIFO+TS *to automaton* VS_RFIFO : SPEC *(presented in Lemma 6.2) is also a refinement mapping from automaton* GCS *to automaton* SELF : SPEC, *under the assumption that clients at each end-point* $\mathtt{p}$ *satisfy the specification automaton* $\mathtt{client} : \mathtt{spec}_p$ *for blocking clients.*

Proof : Automaton SELF : SPEC modifies automaton WV_RFIFO : SPEC by adding a precondition, $\mathtt{last\_dlvrd[p][p]} = \mathtt{LastIndexOf(msgs[p][current\_view[p]])}$, to the steps involving $\mathtt{view}_p()$ actions. We have to show that this precondition is enabled when a step of GCS involving $\mathtt{view}_p(\mathtt{v}, \mathtt{T})$ attempts to simulate a step of SELF : SPEC involving $\mathtt{view}_p(\mathtt{v})$.

Consider the following derivation:

$$\begin{aligned}
\mathtt{s[p].last_dlvrd[p]} &= \max_{r \in T} \mathtt{sync_msg}[r][\mathtt{v.startId}(r)].\mathtt{cut}[p] \text{ (a precondition)} \\
&\geq \mathtt{s[p].sync_msg[p][v.startId(p)].cut[p]} \\
&= \mathtt{s[p].sync_msg[p][s[p].start_change.id].cut[p]} \text{ (a precondition)} \\
&= \mathtt{LastIndexOf(s[p].msgs[p][s[p].current_view])} \text{ (Invariant 6.13).}
\end{aligned}$$

Thus, $\mathtt{R}'(\mathtt{s}).\mathtt{last\_dlvrd}[p][p]) = \mathtt{LastIndexOf}(\mathtt{R}'(\mathtt{s}).\mathtt{msgs}[p][\mathtt{R}'(\mathtt{s}).\mathtt{current\_view}[p]])$ according to mapping $\mathtt{R}'()$, and the precondition is satisfied. ■

Lemma 6.5 establishes function $\mathtt{R}'()$ from Section 6.2 as a refinement mapping from the algorithm automaton GCS to the specification automaton SELF : SPEC. It allows us to conclude the following Theorem.

Theorem 6.4 *Automaton* GCS *implements automaton* SELF : SPEC *in the sense of trace inclusion, under the assumption that clients at each end-point* $\mathtt{p}$ *satisfy the specification automaton* $\mathtt{client} : \mathtt{spec}_p$ *for blocking clients.*

Proof 6.4: Follows from Lemmas 6.1, 6.2, and 6.5. ■

We have shown that GCS implements all of the safety specifications.

Theorem 6.5 *Automaton* GCS *implements each of* WV_RFIFO : SPEC, VS_RFIFO : SPEC, TS : SPEC, *and* SELF : SPEC *automata.*

Proof : Follows from Theorems 6.1, 6.2, 6.3, and 6.4. ■

7 Correctness Proof: Liveness Properties

In this section we prove that our group communication service GCS satisfies the liveness property of Section 4.2. The proof is operational for the most part; In order to show that a certain action is eventually executed, we argue that the preconditions on this action eventually become and stay satisfied and rely on low-level fairness to conclude that the action is eventually executed. The proof also relies on a number of invariants, which we state and prove below.

7.1 Invariants

The following invariant captures the fact that end-points do not deliver messages other than the messages committed in the synchronization messages received from the members of the end-points' transitional sets.

Invariant 7.1 *In every reachable state* $\mathtt{s}$ *of* GCS, *for all* $\mathtt{Proc\ p}$, *if* $\mathtt{s[p].start\_change} \neq \perp$ *and* $\mathtt{s[p].sync\_msg[p][s[p].start\_change.id]} \neq \perp$, *then for all* $\mathtt{Proc\ q} \in \mathtt{s[p].current\_view.set}$,

1. *If* $\mathtt{s[p].start\_change.id} \neq \mathtt{s[p].mbrshp\_view.startId(p)}$, *then*

$$\mathtt{s[p].last\_dlvrd[q]} \leq \mathtt{s[p].sync\_msg[p][s[p].start\_change.id].cut[q]}.$$

2. *Otherwise, let* $\mathtt{v} = \mathtt{s[p].current\_view}$, $\mathtt{v'} = \mathtt{s[p].mbrshp\_view}$, *and let* $\mathtt{T} = \{\mathtt{q} \in \mathtt{v'.set} \cap \mathtt{v.set} \mid \mathtt{sync\_msg[q][v'.startId(q)].view} = \mathtt{v}\}$, *then*

$$\mathtt{s[p].last\_dlvrd[q]} \leq max_{\mathtt{r}\in\mathtt{T}}\ \mathtt{s[p].sync\_msg[r][v'.startId(r)].cut[q]}.$$

Proof 7.1: Both propositions are true in the initial state $\mathtt{s_0}$ since $\mathtt{s_0[p].start\_change} = \perp$. For the inductive step, consider the following critical actions:

<u>$\mathtt{deliver_p(q,m)}$:</u> The proposition remains true because $\mathtt{s'[p].last\_dlvrd[q]} = \mathtt{s[p].last\_dlvrd[q]} + 1$ and because of the precondition on this action, which mimics the statement of this proposition.

<u>MBRSHP.$\mathtt{start\_change_p(id, set)}$:</u> The proposition is vacuously true because $\mathtt{s'[p].sync\_msg[p][id]} = \perp$ (Lemma 6.3).

<u>MBRSHP.$\mathtt{view_p(v)}$:</u> The proposition is true because $\mathtt{p} \in \mathtt{T}$, which follows from Invariant 6.1.

<u>CO_RFIFO.$\mathtt{send_p(set, tag = \textbf{sync\_msg}, cid, v, cut)}$:</u> The proposition is true since $\mathtt{s[p].last\_dlvrd[q]}$ is bounded by $\mathtt{LongestPrefixOf(s[p].msgs[q][s[p].current\_view])}$ in every reachable state of the system for any $\mathtt{Proc\ q} \in \mathtt{s[p].current\_view.set}$ (this fact can be straightforwardly proved by induction) and from the precondition, "$(\forall \mathtt{q} \in \mathtt{s[p].current\_view.set})\ \mathtt{cut(q)} = \mathtt{LongestPrefixOf(s[p].msgs[q][s[p].current\_view])}$", on this action.

<u>CO_RFIFO.$\mathtt{deliver_{q,p}(tag = \textbf{sync\_msg}, cid, v, cut)}$:</u> The proposition is unaffected because the case $\mathtt{q} = \mathtt{p}$ is impossible since end-points don't send cuts to themselves (this can be straightforwardly proved by induction).

<u>$\mathtt{view_p(v, T)}$:</u> The proposition becomes vacuously true because $\mathtt{s'[p].start\_change} = \perp$.

■

The following Invariant states that end-points' cuts specify "real" messages, i.e., those that they do have on their message queues.

Invariant 7.2 *In every reachable state* `s` *of* GCS*, for all* `Proc p`*, if* `s[p].start_change` $\neq \perp$ *and* `s[p].sync_msg[p][s[p].start_change.id]` $\neq \perp$*, then, for all* `Proc q` *and for all* `Int i` *such that* $1 \leq$ `i` $\leq$ `s[p].sync_msg[p][s[p].start_change.id].cut[q]`,

$$\texttt{s[p].msgs[q][s[p].current\_view][i]} \neq \perp.$$

Proof 7.2: The proposition can be straightforwardly proved by induction. The only interesting action is CO_RFIFO.$\texttt{send}_\texttt{p}$(set, tag = **sync_msg**, cid, v, cut). The truth of the proposition after this action is taken follows immediately from the action's precondition: "($\forall$q $\in$ s[p].current_view.set) cut(q) = LongestPrefixOf(s[p].msgs[q][s[p].current_view])." ■

The following Corollary states that if an end-point `p` has end-point `q`'s cut committing certain messages sent by end-point `r` in view `v`, then end-point `q` has those messages buffered.

Corollary 7.1 *In every reachable state* `s` *of* GCS*, for all* `Proc p`, `Proc q`, `Proc r`*, and* `StartChangeId cid`*, it follows that if* `s[p].sync_msg[q][cid]` $\neq \perp$*, then for every integer* `i` *between* 1 *and*
`s[p].sync_msg[q][cid].cut[r]`, `s[q].msgs[r][s[p].sync_msg[q][cid].view][i]` $\neq \perp$.

Proof 7.1: Follows immediately from Invariants 6.7 and 7.2. ■

7.2 Operational Lemmas

Lemma 7.1 *In any execution sequence* α *of* GCS*, the following are true:*

1. *For every* GCS.$\texttt{view}_\texttt{p}$(v, T) *event, there is a preceding* MBRSHP.$\texttt{view}_\texttt{p}$(v) *event such that there is no other* MBRSHP.$\texttt{start\_change}_\texttt{p}$, MBRSHP.$\texttt{view}_\texttt{p}$*, nor* GCS.$\texttt{view}_\texttt{p}$ *events between* MBRSHP.$\texttt{view}_\texttt{p}$(v) *and* GCS.$\texttt{view}_\texttt{p}$(v, T)*.*
2. *For every* MBRSHP.$\texttt{view}_\texttt{p}$(v) *event, there is a preceding* MBRSHP.$\texttt{start\_change}_\texttt{p}$(id, set) *event with* `id = v.startId(p)` *and* `set` $\supseteq$ `v.set`*. Moreover, no* MBRSHP.$\texttt{start\_change}_\texttt{p}$, MBRSHP.$\texttt{view}_\texttt{p}$ *or* GCS.$\texttt{view}_\texttt{p}$ *events occur in* α *between* MBRSHP.$\texttt{start\_change}_\texttt{p}$(id, set) *and* MBRSHP.$\texttt{view}_\texttt{p}$(v)*.*

Proof 7.1:

1. Assume that GCS.$\texttt{view}_\texttt{p}$(v, T) event occurs in α. One of the preconditions on GCS.$\texttt{view}_\texttt{p}$(v, T) is v = p.mbrshp_view, which can only become satisfied as a result of a preceding MBRSHP.$\texttt{view}_\texttt{p}$(v). Also, notice that
   - MBRSHP.$\texttt{start\_change}_\texttt{p}$ event cannot occur between MBRSHP.$\texttt{view}_\texttt{p}$(v) and GCS.$\texttt{view}_\texttt{p}$(v, T) because it would increase the value of p.start_change.id beyond v.startId(p) making it impossible for GCS.$\texttt{view}_\texttt{p}$(v, T) to occur.
   - MBRSHP.$\texttt{view}_\texttt{p}$ event cannot occur between MBRSHP.$\texttt{view}_\texttt{p}$(v) and GCS.$\texttt{view}_\texttt{p}$(v, T) because it would change the value of p.mbrshp_view making it impossible for GCS.$\texttt{view}_\texttt{p}$(v, T) to occur.
   - GCS.$\texttt{view}_\texttt{p}$(v$'$, T) event cannot occur between MBRSHP.$\texttt{view}_\texttt{p}$(v) and GCS.$\texttt{view}_\texttt{p}$(v, T) because it would set p.start_change to $\perp$ making it impossible for GCS.$\texttt{view}_\texttt{p}$(v, T) to occur.

2. Assume that MBRSHP.$\mathtt{view_p(v)}$ event occurs in α. Then MBRSHP.$\mathtt{start\_change_p(id, set)}$ event with $\mathtt{id} = \mathtt{v.startId(p)}$ and $\mathtt{set} \supseteq \mathtt{v.set}$ must precede MBRSHP.$\mathtt{view_p(v)}$ because the initial values for state variables of MBRSHP do not satisfy the preconditions for MBRSHP.$\mathtt{view_p(v)}$, and the only event that can cause these preconditions to become true is MBRSHP.$\mathtt{start\_change_p(id, set)}$.

   Also, notice that

   - MBRSHP.$\mathtt{start\_change_p}$ event cannot occur between MBRSHP.$\mathtt{start\_change_p(id, set)}$ and MBRSHP.$\mathtt{view_p(v)}$ because it would increase the value of MBRSHP.$\mathtt{start\_change[p].id}$ beyond $\mathtt{v.startId(p)}$ making it impossible for MBRSHP.$\mathtt{view_p(v)}$ to occur.
   - MBRSHP.$\mathtt{view_p(v')}$ event cannot occur between MBRSHP.$\mathtt{start\_change_p(id, set)}$ and MBRSHP.$\mathtt{view_p(v)}$ because it would set MBRSHP.$\mathtt{mode[p]}$ to $\mathtt{normal}$ making it impossible for MBRSHP.$\mathtt{view_p(v)}$ to occur.
   - GCS.$\mathtt{view_p(v', T')}$ event cannot occur between MBRSHP.$\mathtt{start\_change_p(id, set)}$ and MBRSHP.$\mathtt{view_p(v)}$ because its two preconditions, $\mathtt{v'} = \mathtt{p.mbrshp\_view}$ and $\mathtt{v'.startId(p)} = \mathtt{p.start\_change.id} = \mathtt{v.startId(p)}$, can not hold together. Indeed, MBRSHP.$\mathtt{view_p(v')}$ is the only action that can satisfy the precondition $\mathtt{v'} = \mathtt{p.mbrshp\_view}$. It can not occur after MBRSHP.$\mathtt{start\_change_p(id, set)}$ for the reasons given above. It also can not occur before MBRSHP.$\mathtt{start\_change_p(id, set)}$ because otherwise it would imply that $\mathtt{v.startId(p)} > \mathtt{v'.startId(p)}$, which would contradict the second precondition.

■

Lemma 7.2 (Liveness) *Let* $\mathtt{v}$ *be a view with* $\mathtt{v.set} = \mathtt{S}$. *Let* α *be a fair execution sequence of a group communication service* GCS *in which, for every* $\mathtt{p} \in \mathtt{S}$, *the action* MBRSHP.$\mathtt{view_p(v)}$ *occurs and is followed by neither* MBRSHP.$\mathtt{view_p}$ *nor* MBRSHP.$\mathtt{start\_change_p}$ *actions. Then at each endpoint* $\mathtt{p} \in \mathtt{S}$, GCS.$\mathtt{view_p(v, *)}$ *eventually occurs. Furthermore, for every* GCS.$\mathtt{send_p(m)}$ *that occurs after* GCS.$\mathtt{view_p(v, *)}$, *and for every* $\mathtt{q} \in \mathtt{S}$, GCS.$\mathtt{deliver_q(p, m)}$ *also occurs.*

Proof 7.2: We first prove that GCS.$\mathtt{view_p(v, *)}$ eventually occurs. Our task is to show, for each $\mathtt{p} \in \mathtt{v.set}$, that action GCS.$\mathtt{view_p(v, *)}$ becomes enabled at some point after $\mathtt{p}$ receives MBRSHP.$\mathtt{view_p(v)}$ and that it stays enabled forever thereafter unless it is executed. The fact that α is a fair execution of GCS then implies that GCS.$\mathtt{view_p(v, *)}$ is in fact executed.

In order for GCS.$\mathtt{view_p(v, *)}$ to become enabled its preconditions on the state variables of $\mathrm{GCS_p}$ must eventually become and stay satisfied:

$$
\begin{array}{l}
\mathtt{v} = \mathtt{p.mbrshp\_view} \\
\mathtt{v.id} \rangle \mathtt{current\_view.id} \\
\mathtt{v.startId(p)} = \mathtt{p.start\_change.id} \\
(\forall\ \mathtt{q} \in \mathtt{v.set} \cap \mathtt{p.current\_view.set})\ \mathtt{p.sync\_msg[q][v.startId(q)]} \neq \perp \\
\mathtt{T} = \{\mathtt{q} \in \mathtt{v.set} \cap \mathtt{p.current\_view.set} \mid \mathtt{p.sync\_msg[q][v.startId(q)].view} = \mathtt{p.current\_view}\} \\
(\forall\ \mathtt{q} \in \mathtt{current\_view.set})\ \mathtt{p.last\_dlvrd[q]} = \max_{\mathtt{r} \in \mathtt{T}} \mathtt{p.sync\_msg[r][v.startId(r)].cut(q)}
\end{array}
$$

We now look at each precondition for GCS.$\mathtt{view_p(v, *)}$ and argue that it eventually becomes satisfied and that it stays satisfied from that point on until GCS.$\mathtt{view_p(v, *)}$ is executed.

$\underline{\mathtt{v} = \mathtt{p.mbrshp\_view} \text{ and } \mathtt{v.id} > \mathtt{current\_view.id}}$: These preconditions become satisfied as a result of MBRSHP.$\mathtt{view_p(v)}$. In particular, notice that the latter becomes satisfies because in any reachable state of the system MBRSHP.$\mathtt{mbrshp\_view} = \mathtt{p.mbrshp\_view} \geq \mathtt{p.current\_view}$ (this can be straightforwardly proved by induction). These preconditions stay satisfied forever, unless GCS.$\mathtt{view_p(v, *)}$ is executed, because, by our assumption, α does not contain any subsequent MBRSHP.$\mathtt{view_p(v')}$, and hence, by contrapositive of Lemma 7.1.1, it also does not contain any subsequent GCS.$\mathtt{view_p(v', *)}$ with $\mathtt{v'} \neq \mathtt{v}$.

$\underline{\mathtt{v.startId(p)} = \mathtt{p.start\_change.id}}$: This condition becomes satisfied as a result of a MBRSHP.$\mathtt{start\_change_p(id, set)}$ event with $\mathtt{id} = \mathtt{v.startId(p)}$ and $\mathtt{set} \supseteq \mathtt{v.set}$, which must precede MBRSHP.$\mathtt{view_p(v)}$ in α by Lemma 7.1.2.

This condition stays satisfied from the time of MBRSHP.$\mathtt{start\_change_p(id, set)}$ at least until GCS.$\mathtt{view_p(v, *)}$ because the only two types of actions, MBRSHP.$\mathtt{start\_change_p(id', set')}$ and GCS.$\mathtt{view_p(v', *)}$ with $\mathtt{v'} \neq \mathtt{v}$ that may affect the value of $\mathtt{p.start\_change}$ cannot occur after MBRSHP.$\mathtt{start\_change_p(id, set)}$, as implied by the assumption on this lemms, Lemma 7.1.2, and the contrapositive of Lemma 7.1.2.

$\underline{(\forall \mathtt{q} \in \mathtt{v.set} \cap \mathtt{p.current\_view.set})\ \mathtt{p.sync\_msg[q][v.startId(q)]} \neq \bot}$: In order to show that $\mathtt{p}$ eventually receives the right synchronization messages from every $\mathtt{q}$ in $\mathtt{v.set} \cap \mathtt{p.current\_view.set}$, we have to prove that (a) $\mathtt{q}$ eventually sends to $\mathtt{p}$ a synchronization message tagged with $\mathtt{v.startId(q)}$ and that (b) CO_RFIFO eventually delivers this message to $\mathtt{p}$.

- We start with part (a). In order for CO_RFIFO.$\mathtt{send_q(set, tag} = \mathbf{sync\_msg}, \mathtt{v.startId(q), v', cut)}$ to happen, the following preconditions have to become and stay satisfied until this action is executed (see the code in Figures 9, 10, and 11):

  1. $\mathtt{start\_change} \neq \bot$;
  2. $\mathtt{start\_change.id} = \mathtt{v.startId(q)}$:
  3. $\mathtt{sync\_msg[p][v.startId(q)]} = \bot$;
  4. $\mathtt{start\_change.set} \subseteq \mathtt{reliable\_set}$;
  5. $\mathtt{block\_status} = \mathtt{blocked}$.

  We now consider each of these preconditions: The first three become true when $\mathtt{q}$ receives MBRSHP.$\mathtt{start\_change_q(v.startId(q), set)}$, the occurrence of which follows from Lemma 7.1.2. This is straightforward for the first two, and is implied by Lemma 6.3 for the third. They remain satisfied from that point on, at least until the synchronization message is sent. This is because, as implied by Lemma 7.1.2 and the assumption on α, there are no subsequent $\mathtt{start\_change}$ events at $\mathtt{q}$ which may set $\mathtt{start\_change.id}$ to a different value, and because the only event (contrapositive of Lemma 7.1.2), GCS.$\mathtt{view_q(v)}$, that can reset $\mathtt{start\_change}$ to $\bot$ may occur only after $\mathtt{q}$ sends the synchronization message.

  The fourth precondition becomes satisfied when CO_RFIFO.$\mathtt{reliable_q(set)}$ occurs with $\mathtt{set} = \mathtt{current\_view.set} \cup \mathtt{start\_change.set}$. This action eventually occurs because it becomes and remains enabled when $\mathtt{q}$ receives MBRSHP.$\mathtt{start\_change_q(v.startId(q), set)}$. Note that although CO_RFIFO.$\mathtt{reliable_q(set)}$ may occur multiple times afterwards, $\mathtt{reliable\_set}$ remains unchanged until GCS.$\mathtt{view_q(v)}$ occurs since both $\mathtt{current\_view.set}$ and $\mathtt{start\_change.set}$ remain unchanged.

  The fifth precondition, $\mathtt{block\_status} = \mathtt{blocked}$, becomes satisfied as a result of a $\mathtt{block\_ok_q()}$ input from the application client at $\mathtt{q}$. Notice that if $\mathtt{block\_status}$ has the value $\mathtt{blocked}$ at anytime after MBRSHP.$\mathtt{start\_change_q(v.startId(q), set)}$ then it does not lose this value until GCS.$\mathtt{view_q(v)}$ happens because neither $\mathtt{block_q()}$ nor $\mathtt{client.block\_ok_q()}$ are enabled after that, and because GCS.$\mathtt{view_q(v)}$ is the only possible GCS view event (the contrapositive of Lemma 7.1.2). To see that block_status does in fact become blocked consider the three possible values of block_status right after MBRSHP.$\mathtt{start\_change_q(v.startId(q), set)}$ occurs:

1. block_status = blocked: We are all set.
2. block_status = requested: By Invariant 6.11, $\mathtt{client.block\_ok_q()}$ is enabled. It stays enabled until it is executed because the actions, $\mathtt{block_q()}$ and GCS.$\mathtt{view_q()}$, which would disable it, cannot occur. When it is executed, the precondition becomes satisfied.
3. block_status = unblocked: When MBRSHP.$\mathtt{start\_change_q(v.startId(q), set)}$ occurs, $\mathtt{block_q()}$ becomes and stays enabled until it is executed. After that, block_status becomes requested and the same reasoning as in the previous case applies.

Since all the preconditions for CO_RFIFO.$\mathtt{send_q(set, tag = \textbf{sync\_msg}, v.startId(q), v', cut)}$ at some point become and stay satisfied, this action is executed and the synchronization message is appended to CO_RFIFO.channel[q][p].

- For part (b), we have to show that CO_RFIFO eventually delivers this synchronization message to p. For this we have to argue that, when the synchronization message is placed on CO_RFIFO.channel[q][p] and at least until it is delivered to p, end-point p is in both CO_RFIFO.reliable_set[q] and CO_RFIFO.live_set[q]. The former implies that CO_RFIFO does not lose any messages (in particular, this synchronization message) from q to p. It becomes satisfied because, as was argued above, CO_RFIFO.$\mathtt{reliable_q(set)}$ with set = current_view.set $\cup$ start_change.set eventually occurs, and because

$$\mathtt{p} \in \mathtt{v.set} \subseteq \mathtt{q.start\_change.set} \subseteq \mathtt{set} = \text{CO\_RFIFO}\mathtt{.reliable\_set[q]}.$$

  This precondition remains satisfied afterwards because CO_RFIFO.reliable_set[q] remains unchanged until GCS.$\mathtt{view_q(v, *)}$ occurs (as was explained above). The latter, in conjunction with low-level fairness, implies that CO_RFIFO eventually delivers every message (in particular, this synchronization message) on the channel from q to p. It becomes satisfied because MBRSHP.$\mathtt{view_q(v)}$ is linked to CO_RFIFO.$\mathtt{live\_set_q(v.set)}$ and because $\mathtt{p} \in \mathtt{v.set}$. This precondition remains satisfied afterward because α does not contain any subsequent MBRSHP events at end-point q.

Thus, end-point p eventually receives the right synchronization messages from every q in v.set $\cap$ p.current_view.set.

<u>$(\forall \mathtt{q} \in \mathtt{current\_view.set})$ $\mathtt{p.last\_dlvrd[q]} = \mathtt{max_{r \in T} p.sync\_msg[r][v.startId(r)].cut[q]}$:</u>

By Invariant 7.1, p.last_dlvrd[q] never exceeds $\mathtt{max_{r \in T}}$ $\{\mathtt{p.sync\_msg[r][v.startId(r)].cut[q]}\}$ at any q. It is therefore left to show that p.last_dlvrd[q] does not remain smaller than $\mathtt{max_{r \in T}}$.

We have shown above that all the other preconditions for delivering view v by p eventually become and remain satisfied until the view is delivered. Consider the part of α after all of these preconditions hold. Let q be an end-point in current_view.set such that $\mathtt{p.last\_dlvrd[q]} < \mathtt{max_{r \in T} p.sync\_msg[r][v.startId(r)].cut[q]}$. Let $\mathtt{i} = \mathtt{p.last\_dlvrd[q]} + 1$. We now argue that p.last_dlvrd[q] eventually becomes i, i.e., that p eventually delivers the next message from q. Applying this argument inductively, implies that p.last_dlvrd[q] eventually reaches $\mathtt{max_{r \in T}}$ $\{\mathtt{p.sync\_msg[r][v.startId(r)].cut[q]}\}$.

Notice that all the preconditions, except perhaps p.msgs[q][p.current_view][i] $\neq \perp$, for delivering the i'th message from q are satisfied because they are the same as the preconditions on p delivering view v that we have shown to be satisfied. Thus, if the i'th message is already on p.msgs[q][p.current_view][i], then delivery of this message eventually occurs by low-level fairness, p.last_dlvrd[q] is incremented, and we are all done.

Therefore, consider the case when p lacks the i'th message from q.

There are two possibilities:

If end-point q is in p's transitional set T for view v, then we know the following:

- q's view prior to installing view v is the same as p's current view (by definition of T and Invariant 6.9).
- q's reliable_set contains p starting before q sent any messages in that view and continuing for the rest of α.
- Corollary 7.1 implies that r has this message and all the messages that precede it in r.msgs[r][p.current_view].
- End-point r is enabled to send these messages to p in FIFO order. The only event that could prevent r from sending these messages is $\texttt{gcs.view}_{\texttt{r}}(\texttt{v})$, as it would change the value of r.current_view. However, Self Delivery implies that this event cannot happen until r self-delivers its own messages, which in turn is preconditioned on r sending these messages to others via CO_RFIFOchannels.
- The fact that the connection between q and p is live at least after q receives MBRSHP.$\texttt{view}_{\texttt{q}}(\texttt{v})$ implies that CO_RFIFO eventually delivers this message to p.

Otherwise, if end-point q is not in p's transitional set T for view v, we know by the fact that i is $\leq \max_{r \in T}$ {p.sync_msg[r][v.startId(r)].cut[q]}, that there exist some end-points in T whose synchronization messages commit them to delivering the i'th message from q in view p.current_view. Let r be an end-point with a smallest identifier among these end-points. Here is what we know:

- Corollary 7.1 implies that r has this message on its r.msgs[r][p.current_view] queue.
- q's reliable_set contains p starting before q sent any messages in that view and continuing for the rest of α.
- Upon examination of each of the ForwardingStrategyPredicates in Section 5.2.2, we see that the preconditions for r forwarding the i'th message of q to a set including p eventually become and stay satisfied.
- Since in both forwarding strategies there is a finite number of messages from q sent in this view that can be forwarded, the low-level fairness condition implies that the i's message is eventually forwarded to p.
- The fact that the connection between q and p is live at least after q receives MBRSHP.$\texttt{view}_{\texttt{q}}(\texttt{v})$ implies that CO_RFIFO eventually delivers this message to p.

Therefore, the i'th message from q is eventually delivered to p, and since the preconditions on delivering this message to the application client at p are satisfied, this delivery eventually occurs and p.last_dlvrd[q] is incremented. Applying this argument inductively, we conclude that p.last_dlvrd[q] eventually reaches $\max_{r \in T}$p.sync_msg[r][v.startId(r)].cut[q] for every q in current_view.set).

We have shown that each precondition on $\mathtt{p}$ delivering $\textsc{gcs}.\mathtt{view_p}(\mathtt{v},*)$ eventually becomes and stays satisfied. Low-level fairness implies that $\textsc{gcs}.\mathtt{view_p}(\mathtt{v},*)$ eventually occurs.

Now, consider the second part of the theorem. We have to prove that after $\textsc{gcs}.\mathtt{view_p}(\mathtt{v},*)$ occurs at $\mathtt{p}$, for every $\textsc{gcs}.\mathtt{send_p}(\mathtt{m})$ event that occurs at $\mathtt{p}$, there is a corresponding event $\textsc{gcs}.\mathtt{deliver_q}(\mathtt{p},\mathtt{m})$ that occurs at $\mathtt{q} \in \mathtt{v.set}$. Consider the following argument:

1. For the rest of α, after $\textsc{gcs}.\mathtt{view_p}(\mathtt{v},*)$ occurs, $\textsc{co\_rfifo}.\mathtt{live\_set}[\mathtt{p}]$ is equal to $\mathtt{v.set}$.

   This is true because $\textsc{co\_rfifo}.\mathtt{live\_set}[\mathtt{p}]$ is set to $\mathtt{v.set}$ when $\textsc{mbrshp}.\mathtt{view_p}(\mathtt{v})$ occurs and remains unchanged thereafter because of the assumption that α does not contain any subsequent MBRSHP events at end-point $\mathtt{p}$.

2. After $\textsc{gcs}.\mathtt{view_p}(\mathtt{v},*)$ occurs and before any $\textsc{co\_rfifo}.\mathtt{send_p}$ event involving a ViewMsg or an AppMsg occurs, $\mathtt{p}$ eventually executes $\textsc{co\_rfifo}.\mathtt{reliable_p}(\mathtt{v.set})$. Moreover, after that and forever thereafter, both $\mathtt{p.reliable\_set}$ and $\textsc{co\_rfifo}.\mathtt{reliable\_set}[\mathtt{p}]$ equal $\mathtt{v.set}$.

   This is true because $\textsc{gcs}.\mathtt{view_p}(\mathtt{v},*)$ sets $\mathtt{p.start\_change}$ to $\perp$ and $\mathtt{p.current\_view.set}$ to $\mathtt{v.set}$, thus enabling $\textsc{co\_rfifo}.\mathtt{reliable_p}(\mathtt{v.set})$. This action is eventually happens because of low-level fairness and the facts that for the rest of α there are no subsequent $\mathtt{start\_change_p}$ and $\textsc{gcs}.\mathtt{view_p}(\mathtt{v}',*)$ events. Moreover, since $\mathtt{p.start\_change}$ and $\mathtt{p.current\_view.set}$ remain unchanged because of the latter reason, whenever $\textsc{co\_rfifo}.\mathtt{reliable_p}$ subsequently happens, both $\mathtt{p.reliable\_set}$ and $\textsc{co\_rfifo}.\mathtt{reliable\_set}[\mathtt{p}]$ remain equal to $\mathtt{v.set}$.

   From the above two points and low-level fairness, it follows that, whatever messages $\mathtt{p}$ sends to $\mathtt{q}$ afterwards, it will be eventually delivered to $\mathtt{q}$. The arguments below about delivery of messages to $\mathtt{q}$ rely on this fact.

3. After $\textsc{co\_rfifo}.\mathtt{reliable_p}(\mathtt{v.set})$ occurs, $\textsc{co\_rfifo}.\mathtt{send_p}(\mathtt{v.set}-\{\mathtt{p}\},\mathtt{tag}=\mathbf{view\_msg},\mathtt{v})$ eventually occurs. Moreover, after that $\mathtt{p}$ does not send any ViewMsg in the future,

   This follows directly from low-level fairness and the code in Figure 10.

4. Every process $\mathtt{q} \in \mathtt{v.set}\text{-}\{\mathtt{p}\}$ eventually receives a ViewMsg containing $\mathtt{v}$ from $\mathtt{p}$. At that time it sets its q.view_msg[p] to v.

5. After q receives the ViewMsg from p, q.view_msg[p] remains set to v for the remainder of α.

   This is true because, as was argued before, p sends only one view_msg since the time it moves in to view v.

6. Moreover, the ViewMsg q receives from p is received prior to q receiving any application message sent by p in view v.

   This is true because p sends application messages in view v only after sent the ViewMsg, and because CO_RFIFO preserves the order in which messages are sent.

7. After p has sent the ViewMsg, for the rest of α, if $\mathtt{p.msgs}[\mathtt{p}][\mathtt{v}][\mathtt{p.last\_sent}+1]$ contains a message (say m), $\textsc{co\_rfifo}.\mathtt{send_p}(\mathtt{v.set}-\{\mathtt{p}\},\mathtt{tag}=\mathbf{app\_msg},\mathtt{m})$ is enabled and hence eventually occurs by low-level fairness.

8. Message m is eventually delivered to every $\mathtt{q} \in \mathtt{v.set}\text{-}\{\mathtt{p}\}$.

9. Since, as was argued above, q.view_msg[p] is equal to v at the time q receives m, q appends m to its q.msgs[p][v] queue.

10. Since `p.last_sent` is incremented after each application message is sent, any message on $\texttt{p.msgs}[\texttt{p}][\texttt{v}]$ is eventually sent out and is delivered to each $\texttt{q} \in \texttt{v.set-}\{\texttt{p}\}$.

11. Once `q` moves in to view `v`, the delivery of a message $\texttt{m}'$ from `q.msgs[p][v][q.last_dlvrd+1]` becomes enabled and is thus eventually executed by low-level fairness. If $\texttt{p} = \texttt{q}$, there is an additional precondition that requires `q` to first send $\texttt{m}'$ to others in `v` via CO_RFIFO, before it can self-deliver $\texttt{m}'$ to its application client. This precondition becomes eventually satisfied, as was argued above.

12. Since `q.last_dlvrd[p]` is incremented after each application message is delivered to `q`'s application client, any message on $\texttt{q.msgs}[\texttt{p}][\texttt{v}]$ is eventually delivered.

Therefore, taking everything said above in to consideration, when the application client at `p` sends a message `m` in view `v`, `m` is eventually delivered to the application client at every $\texttt{q} \in \texttt{v.set}$. ■

8 Modeling Crash and Recovery of End-Points

In this section we show that the service presented in Section 5 also provides meaningful and correct semantics in the environment where GCS end-points can crash and recover. In particular, it allows the recovered GCS end-points to continue running the algorithm under their original identity (in contrast e.g., to [12] which requires recovered processes to assume new identities). Furthermore, GCS end-points do not need to store any information on stable storage: upon recovery, they can re-start the algorithm with their variables in initial state.

In order to model crash and recovery in our algorithms, the input actions $\texttt{crash}_\texttt{p}()$ and $\texttt{recover}_\texttt{p}()$ are added to all the automata running at location `p`; these include the application clients, the group communication service end-points, the reliable connection-oriented FIFO service and the membership service. We view the $\texttt{crash}_\texttt{p}()$ as causing the application clients and the group communication service end-points to crash, and notifying the membership service of the crash. The membership service does not crash and does not lose it's state. $\texttt{recover}_\texttt{p}()$ sets $\texttt{mbrshp.mode}[\texttt{p}]$ to normal.

A boolean flag $\texttt{crashed}_\texttt{p}$, initially set to `false`, is added to the state of the application client and the group communication service end-points at location `p`. The $\texttt{crash}_\texttt{p}()$ input sets this flag to `true`. When $\texttt{crashed}_\texttt{p}$ is `true`, all locally controlled actions and the effects of all input actions of both the group communication end-point and its application client at process `p` are disabled. The $\texttt{recover}_\texttt{p}()$ action resets all the state variables, including the $\texttt{crashed}_\texttt{p}$ flag, to their initial values.

Notice that, recovering to an initial default view does not violate Local Monotonicity as this initial view is not delivered to the application. The first view delivered after a recovery is guaranteed to satisfy Local Monotonicity since the membership service does not crash and hence maintains the last $\texttt{startchange}[\texttt{p}]\texttt{.id}$ and $\texttt{mbrshp\_view}[\texttt{p}]\texttt{.id}$ values.

The connection-oriented FIFO service responds to $\texttt{crash}_\texttt{p}()$ by making both $\texttt{reliable\_set}[\texttt{p}]$ and $\texttt{live\_set}[\texttt{p}]$ empty, thus allowing the last messages from the crashed `p` to other processes to be dropped.

The specifications of Section 4 are adapted to account for crash and recovery in a similar manner. They respond to $\texttt{crash}_\texttt{p}()$ and $\texttt{recover}_\texttt{p}()$ actions in the same way as the algorithms do, except to preserve the pre-crashed values of the `start_change` and `current_view` variables upon recovery.

This specifies that the recovered processes still have to preserve Local Monotonicity on start_change and view identifiers.

It is possible to show that all the invariants and simulations proven in Section 6 still hold whenever $\texttt{crashed}_{\texttt{p}}$ is `false`. In order to formally prove these invariants and simulations, one needs to use history variables which recall past values of process states before their crashes. Providing such formal proofs is beyond the scope of this paper.

9 Conclusions

We have designed a novel group multicast service that interacts with an external membership service to provide virtually synchronous communication semantics. Our service is currently being implemented as part of a novel architecture for scalable group communication in WANs. We have constructed a virtually synchronous group multicast algorithm which exchanges one round of synchronization messages during reconfiguration, in parallel with the execution of a group membership algorithm. In contrast to previously suggested virtual synchrony algorithms, (e.g., [7, 22, 16, 5, 33]) our algorithm does not require processes to conduct an additional communication round in order to pre-agree upon a globally unique identifier.

This feature is achieved by virtue of a simple yet powerful idea: the inclusion of a list of *locally* unique start_change identifiers in the membership view. The inclusion of such identifiers in the view eliminates the need to tag messages with a common (globally unique) identifier.

Two views are considered the same only if they carry the same list of start_change identifiers. The membership service of [27] which we use in our implementation guarantees that if the network eventually stabilizes, all the processes eventually receive the *same* `view` (with the same list of start_change identifiers) and do not receive new `view` or `start_change` messages henceforward. In such cases, the entire group communication service is live: the last membership view is delivered to the application, as well as all the messages sent in this view.

The start_change interface is an important aspect of the design of a client-server oriented group communication service which decouples membership maintenance from group multicast in order to provide scalable group membership services in WANs. Maestro [11] also separates the maintenance of membership from group multicast. Unlike Maestro [11], in our design, the client does not wait for the membership to agree upon a globally unique identifier before starting the virtual synchrony algorithm, and the membership service does not wait for responses from clients asserting that virtual synchrony was achieved before delivering views.

We have implemented the virtually synchronous group multicast service and are currently testing its scalability limits. In order to increase the scalability, we intend to explore ways to incorporate a two-tier hierarchy into our algorithm, as suggested by Guo et al. [22]. With this approach, processes will not directly send cut messages to all of their peers. Instead, messages will be sent by each process to its designated leader, which will in turn, aggregate the cut messages into a single message and forward it to the other leaders. We have presented the algorithm at an abstract level that would allow incorporating such extentions without violating its correctness.

Acknowledgments

We thank Nancy Lynch, Alex Shvartsman and Jeremy Sussman for many discussions and helpful suggestions.

References

[1] ACM. *Commun. ACM 39(4), special issue on Group Communications Systems*, April 1996.

[2] O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France*, June 1993. LNCS 774.

[3] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *3rd International Workshop on Services in Distributed and Networked Environment (SDNE)*, pages 84–91, June 1996.

[4] Y. Amir, G. V. Chokler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 183–192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997. Full version available as Technical Report CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[5] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS)*, July 1992.

[6] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[7] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4), November 1995.

[8] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In M. Mavronicolas, M. Merritt, and N. Shavit, editors, *Networks in Distributed Computing (DIMACS workshop)*, volume 45 of *DIMACS*, pages 23–42. American Mathematical Society, 1998.

[9] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 244–252, June 1999.

[10] Ö. Babaoğlu, R. Davoli, and A. Montresor. Partitionalbe Group Membership: Specification and Algorithms. TR UBLCS97-1, Department of Conmputer Science, University of Bologna, January 1997.

[11] K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware support for distributed multimedia and collaborative computing. In *Multimedia Computing and Networking (MMCN98)*, 1998.

[12] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[13] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 237–246, June 1998.

[14] F. Cristian and F. Schmuck. Agreeing on Process Group Membership in Asynchronous Distributed Systems. Technical Report CSE95-428, Department of Computer Science and Engineering, University of California, San Diego, 1995.

[15] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 227–236, June 1998.

[16] D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. Technical Report CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[17] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[18] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62, August 1997.

[19] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. TR 95-1537, dept. of Computer Science, Cornell University, August 1995.

[20] R. Friedman and A. Vaysburg. Fast replicated state machines over partitionable networks. In *16th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 1997.

[21] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, Oct. 1997. IEEE Computer Society Press.

[22] K. Guo, W. Vogels, and R. van Renesse. Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication. In *7th ACM SIGOPS European Workshop*, September 1996.

[23] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for ensemble layers. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer-Verlag, Mar. 1999.

[24] M. Hiltunen and R. Schlichting. Properties of membership services. In *2nd International Symposium on Autonomous Decentralized Systems*, pages 200–207, 1995.

[25] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[26] I. Keidar, R. Khazan, N. Lynch, and A. Shvartsman. Extending I/O automaton methodology for modeling complex systems, (generalized composition and inheritance). Technical report, MIT Laboratory for Computer Science, 1999. In preparation.

[27] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. Technical Report CS99-623, Department of Conmputer Science and Engineering, University of California, San Diego, June 1999. Also MIT Technical Memorandum MIT-LCS-TM-593.

[28] R. Khazan, A. Fekete, and N. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on DIStributed Computing (DISC)*, pages 258–272, Andros, Greece, September 1998.

[29] B. Lampson. Generalizing Abstraction Functions. Massachusetts Institute of Technology, Laboratory for Computer Science, principles of computer systems class, handout 8, 1997. ftp://theory.lcs.mit.edu/pub/classes/6.826/www/6.826-top.html.

[30] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[31] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[32] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994. Full version: technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[33] O. Rodeh. The Design and Implementation of Lansis/E. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, May 1997.

[34] A. Schiper and A. Ricciardi. Virtually synchronous communication based on a weak failure suspector. *Digest of Papers, FTCS-23*, pages 534–543, June 93.

[35] F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[36] I. Shnaiderman. Implementation of Reliable Datagram Service in the LAN environment. Lab project, The Hebrew University of Jerusalem, January 1999. http://www.cs.huji.ac.il/~transis/publications.html.

[37] J. Sussman and K. Marzullo. The *Bancomat* problem: An example of resource allocation in a partitionable asynchronous system. In *12th International Symposium on DIStributed Computing (DISC)*, September 1998. Full version: Tech Report 98-570 University of California, San Diego Department of Computer Science and Engineering.

[38] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Commun. ACM*, 39(4), April 1996.

[39] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report CS99-31, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, September 1999. Also Technical Report MIT-LCS-TR-790, Massachusetts Institute of Technology, Laboratory for Computer Science and Technical Report CS0964, Computer Science Department, the Technion, Haifa, Israel.

A Review of Proof Techniques

In this section we describe the main techniques used to prove correctness of I/O automata: invariant assertions, hierarchical proofs, refinement mappings, and history and prophecy variables. The material in this section is closely based on [30, pages 216-228] and [29, pages 3,4, and 13]. In Section A.3 we present a proof-extension theorem of [26] that provides a formal framework for the reuse of simulation proofs based on refinement mappings.

A.1 Invariants

The most fundamental type of property to be proved about an automaton is an *invariant assertion*, or just *invariant*, for short. An invariant assertion of an automaton A is defined as any property that is true in every single reachable state of A.

Invariants are typically proved by induction on the number of steps in an execution leading to the state in question. While proving an inductive step, we consider only *critical actions*, which affect the state variables appearing in the invariant.

A.2 Hierarchical Proofs

One of the important proof strategies is based on a hierarchy of automata. This hierarchy represents a series of descriptions of a system or algorithm, at different levels of abstraction. The process of moving through the series of abstractions, from the highest level to the lowest level, is known as *successive refinement*. The top level may be nothing more than a problem specification written in the form of an automaton. The next level is typically a very abstract representation of the system: it may be centralized rather than distributed, or have actions with large granularity, or have simple but inefficient data structures. Lower levels in the hierarchy look more and more like the actual system or algorithm that will be used in practice: they may be more distributed, have actions with small granularity, and contain optimizations. Because of all this extra detail, lower levels in the hierarchy are usually harder to understand than the higher levels. The best way to prove properties of the lower-level automata is by relating these automata to automata at higher levels in the hierarchy, rather than by carrying out direct proofs from scratch.

A.2.1 Refinement Mappings

The simplest way to relate two automata, say A and S, is to present a *refinement mapping* R from the reachable states of A to the reachable state of S such that it satisfies the following two conditions:

1. If t_0 is an initial state of A, then $R(s_0)$ is an initial state of S.
2. If t and $R(t)$ are reachable states of A and S respectively, and (t, π, t') is a step of A, then there exists an execution fragment of S beginning at state $R(t)$ and ending at state $R(t)'$, with its trace being the same as the trace of π and its final state $R(t)'$ being the same as $R(t')$.

The first condition asserts that any initial state of A has some corresponding initial state of S. The second condition asserts that any step of A has a corresponding sequence of steps of S. This corresponding sequence can consist of one step, many steps, or even no steps, as long as the correspondence between the states is preserved and the external behavior is the same.

The following theorem gives the key property of refinement mappings:

Theorem A.1 *If there is a refinement mapping from A to S, then*

$$\texttt{traces}(A) \subseteq \texttt{traces}(S).$$

If automata A and S have the same external signature and the traces of A are the traces of S, then we say that A *implements* S *in the sense of trace inclusion*, which means that A never does anything that S couldn't do. Theorem A.1 implies that, in order to prove that one automaton implements another in the sense of trace inclusion, it is enough to produce a refinement mapping from the former to the latter.

A.2.2 History and Prophecy Variables

Sometimes, however, even when the traces of A are the traces of S, it is not possible to give a refinement mapping from A to S. This may happen due to the following two generic reasons:

- The states of S may contain more information than the states of A.
- S may make some premature choices, which A makes later.

The situation when A has been optimized not to retain certain information that S maintains can be resolved by augmenting the state of A with additional components, called *history variables* (because they keep track of additional information about the history of execution), subject to the following constraints:

1. Every initial state has at least one value for the history variables.
2. No existing step is disabled by the addition of predicates involving history variables.
3. A value assigned to an existing state component must not depend on the value of a history variable.

These constraints guarantee that the history variables simply record additional state information and do not otherwise affect the behavior exhibited by the automaton. If the automaton A_{HV} augmented with history variables can be shown to implement S by presenting a refinement mapping, it follows that the original automaton A without the history variables also implements S, because they have the same traces.

The situation when S is making a premature choice, which A makes later, can be resolved by augmenting A with a different sort of auxiliary variable, *prophecy variable*, which can look into the future just as history variable looks into the past. A prophecy variable guesses in advance some non-deterministic choice that A is going to make later. The guess gives enough information to construct a refinement mapping to S (which is making the premature choice). For an added variable to be a prophecy variable, it must satisfy the following conditions:

1. Every state has at least one value for the prophecy variable.
2. No existing step is disabled *in the backward direction* by the new preconditions involving a prophecy variable. More precisely, for each step (t, π, t') there must be a state (t, p) and a p' such that there is a step $((t, p), \pi, (t', p'))$.
3. A value assigned to an existing state component must not depend on the value of prophecy variable.

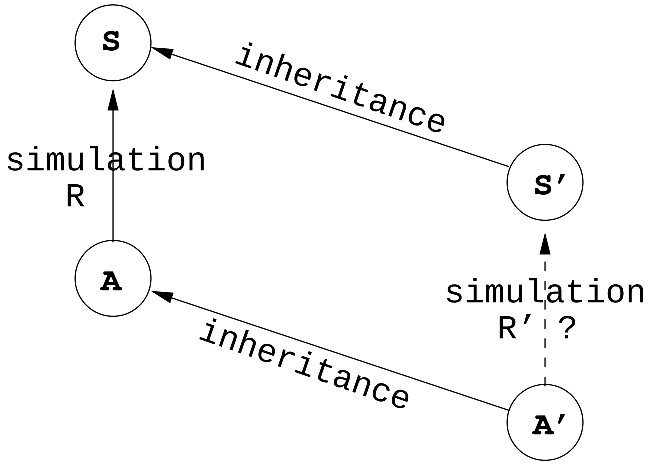
4. If t is an initial state of A and (t, p) is a state of the A augmented with the prophecy variable, then it must be its initial state.

If these conditions are satisfied, the automaton augmented with the prophecy variable will have the same (finite) traces as the automaton without it. Therefore, if we can exhibit a refinement mapping from A_{PV} to S, we know that the A implements S.

A.3 Inheritance and Proof Extension Theorem

We now present a theorem from [26] which lays the foundation for incremental proof construction. Consider the example illustrated in Figure 13, where a refinement mapping $\mathtt{R}$ from an algorithm $\mathtt{A}$ to a specification $\mathtt{S}$ is given, and we want to construct a refinement mapping $\mathtt{R}'$ from a child $\mathtt{A}'$ of an automaton $\mathtt{A}$ to a child $\mathtt{S}'$ of a specification automaton $\mathtt{S}$.

Figure 13 Algorithm $\mathtt{A}$ simulates specification $\mathtt{S}$ with $\mathtt{R}$. Can $\mathtt{R}$ be reused for building a refinement $\mathtt{R}'$ from a child $\mathtt{A}'$ of $\mathtt{A}$ to a child $\mathtt{S}'$ of $\mathtt{S}$?



Theorem A.2 below implies that such a refinement $\mathtt{R}'$ can be constructed by supplementing $\mathtt{R}$ with a mapping $\mathtt{R_n}$ from the states of $\mathtt{A}'$ to the state extension introduced by $\mathtt{S}'$. Mapping $\mathtt{R_n}$ has to map every initial state of $\mathtt{A}'$ to some initial state extension of $\mathtt{A}'$ and it has to satisfy a step condition similar to the one for refinement mapping (Sec. A.2.1), but only involving the transition restriction of $\mathtt{S}'$.

Theorem A.2 *Let automaton $\mathtt{A}'$ be a child of automaton $\mathtt{A}$. Let automaton $\mathtt{S}'$ be a child of automaton $\mathtt{S}$. Let mapping $\mathtt{R}$ be a refinement from $\mathtt{A}$ to $\mathtt{S}$.*

Let $\mathtt{R_n}$ be a mapping from the states of $\mathtt{A}'$ to the state extension introduced by $\mathtt{S}'$.

A mapping $\mathtt{R}'$ from the states of $\mathtt{A}'$ to the states of $\mathtt{S}'$, defined in terms of $\mathtt{R}$ and $\mathtt{R_n}$ as

$$\mathtt{R}'(\langle \mathtt{t}, \mathtt{t_n} \rangle) = \langle \mathtt{R}(\mathtt{t}), \mathtt{R_n}(\langle \mathtt{t}, \mathtt{t_n} \rangle) \rangle$$

is a refinement *from $\mathtt{A}'$ to $\mathtt{S}'$ if $\mathtt{R}'$ satisfies the following two conditions:*

1. *If $\mathtt{t}$ is an initial state of $\mathtt{A}'$, then $\mathtt{R_n}(\mathtt{t})$ is an initial state extension of $\mathtt{S}'$.*
2. *If $\langle \mathtt{t}, \mathtt{t_n} \rangle$ is a reachable state of $\mathtt{A}'$, $\mathtt{s} = \langle \mathtt{R}(\mathtt{t}), \mathtt{R_n}(\langle \mathtt{t}, \mathtt{t_n} \rangle) \rangle$ is a reachable state of $\mathtt{S}'$, and $(\langle \mathtt{t}, \mathtt{t_n} \rangle, \pi, \langle \mathtt{t}', \mathtt{t}'_\mathtt{n} \rangle)$ is a step of $\mathtt{A}'$, then there exists a finite sequence α of alternating states and actions of $\mathtt{S}'$, beginning from $\mathtt{s}$ and ending at some state $\mathtt{s}'$, and satisfying the following conditions:*

(a) α *projected onto states of* $\mathtt{S}$ *is an execution sequence of* $\mathtt{S}$.

(b) Every step $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}})$ *in* α *is consistent with the transition restriction that* $\mathtt{S}'$ *places on* $\mathtt{S}$.

(c) The parent component of the final state $\mathtt{s}'$ *is* $\mathtt{R}(\mathtt{t}')$.

(d) The child component of the final state $\mathtt{s}'$ *is* $\mathtt{R_n}(\langle \mathtt{t}', \mathtt{t}'_\mathtt{n} \rangle)$.

(e) α *has the same trace as* π.

In practice, one would exploit this theorem as follows: The simulation proof between the parent automata already provides a corresponding execution sequence of the parent specification for every step of the parent algorithm. It is typically the case that the same execution sequence, padded with new state variables, corresponds to the same step at the child algorithm. Thus, conditions 2a, 2c, and 2e of Theorem A.2 hold for this sequence. The only conditions that have to be checked are 2b, and 2d, i.e., that every step of this execution sequence is consistent with the transition restriction placed on $\mathtt{S}$ by $\mathtt{S}'$ and that the values of the new state variables of $\mathtt{S}'$ in the final state of this execution match those obtained when $\mathtt{R_n}$ is applied to the post-state of the child algorithm.

A.4 Safety versus Liveness

Proving that one automaton implements another in the sense of trace inclusion constitutes only *partial correctness*, as it implies *safety* but not *liveness*. In other words, partial correctness ensures than "bad" things never happen, but it does not say anything whether some "good" thing eventually happens.

In this paper, we use invariant assertions and simulation techniques to prove that our algorithms satisfy safety properties, which are stated as I/O automata. For liveness proofs, we use a combination of invariant assertions and carefully proven operational arguments.