SIAM J. COMPUT.
Vol. 32, No. 1, pp. 78–130

A VIRTUALLY SYNCHRONOUS GROUP MULTICAST ALGORITHM FOR WANS: FORMAL APPROACH*

IDIT KEIDAR† AND ROGER KHAZAN‡

Abstract. This paper presents a formal design for a novel group communication service targeted for wide-area networks (WANs). The service provides virtual synchrony semantics. Such semantics facilitate the design of fault tolerant distributed applications. The presented design is more suitable for WANs than previously suggested ones. In particular, it features the first algorithm to achieve virtual synchrony semantics in a single communication round. The design also employs a scalable WAN-oriented architecture: it effectively decouples the main two components of virtually synchronous group communication—group membership and reliable group multicast. The design is carried out formally and rigorously. This paper includes formal specifications of both safety and liveness properties. The algorithm is formally modeled and assertionally verified.

Key words. group communication, virtual synchrony, reliable multicast, formal modeling

AMS subject classifications. 68M14, 68M15, 68W15, 68Q85

PII. S0097539700378754

1. Introduction.

Group communication services (GCSs) [1, 10] are powerful middleware systems that facilitate the development of fault tolerant distributed applications. These services provide a notion of *group abstraction*, which allows application processes to easily organize themselves into multicast groups. Application processes can communicate with the members of a group by addressing messages to the group. Most GCSs strive to present different members of the same group with mutually consistent perceptions of the communication done in the group. This perception is known as *virtual synchrony* (VS) semantics [12].

Traditionally, GCSs were designed for deployment in local-area networks (LANs). Efficient GCSs that operate in wide-area networks (WANs) is still an open area of research. Designing such GCSs is challenging because in WANs communication is more expensive and connectivity is less stable than in LANs.

In this paper we present a novel algorithm for a GCS targeted for WANs. The service provided by our GCS satisfies a variant of the VS semantics that has been shown to be useful for facilitating the design of distributed applications [16, 10]. Our algorithm for implementing this semantics is more appropriate for WANs than the existing solutions: it requires fewer rounds of communication and is designed for the scalable WAN-oriented architecture of [6, 31]. Our design is carried out at a very high level of formality and rigor, much higher than that of most previous designs of virtually synchronous GCSs. It includes formal and precise specifications, algorithms,

*Received by the editors September 27, 2000; accepted for publication (in revised form) June 25, 2002; published electronically November 19, 2002. This paper is an extended version of A client-server approach to virtually synchronous group multicast: Specifications and algorithms, 20th International Conference on Distributed Computing Systems (ICDCS), 2000, IEEE, pp. 344–355. This work was supported by Air Force Aerospace Research (OSR) grants F49620-00-1-0097 and F49620-00-1-0327, Nippon Telegraph and Telephone (NTT) grant MIT9904-12, and NSF grants CCR-9909114 and EIA-9901592.

http://www.siam.org/journals/sicomp/32-1/37875.html

†Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square, Cambridge, MA 02139. Current address: Department of Electrical Engineering, Technion - Israel Institute of Technology, Technion City, Haifa, 32000 Israel (idish@ee.technion.ac.il).

‡Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square, Cambridge, MA 02139 (roger@lcs.mit.edu).

and proofs.

The rest of this section is organized as follows: In section 1.1 we present some basic background on GCSs and VS. Section 1.2 summarizes the contributions made by our work, and section 1.3 gives a roadmap to the rest of the paper.

1.1. Background. Modern distributed applications often involve large groups of geographically distributed processes that interact by sending messages over an asynchronous fault-prone network. Many of these applications maintain a replicated state of some sort. In order for these applications to be correct, the replicas must remain mutually consistent throughout the execution of the application. For example, in an online game, the states of the game maintained by different players must be mutually consistent in order for the game to be meaningful to the players. Designing algorithms that maintain state consistency is difficult however: different application processes may perceive the execution of the application inconsistently because of asynchrony and failures. For example, if Alice, Bob, and Carol are playing an online game, the following asymmetric scenario is possible: Alice and Bob perceive each other as alive and well, but they differ in the way they perceive Carol; one sees Carol as crashed or disconnected, while the other sees her as alive and well. Middleware systems that hide from the application some of the underlying inconsistencies and instead present them with a more consistent picture of the distributed execution facilitate development of distributed applications.

GCSs, such as [3, 5, 44, 12], are examples of such middleware systems. They are particularly useful for building applications that require reliable multipoint to multipoint communication among a group (or groups) of processes. Examples of such applications are data replication (for example, [29, 4, 22, 33, 24]), highly available servers (for example, [8]), and online games. GCSs allow application processes to organize themselves easily into groups and to communicate with all the members of a group by addressing messages to the group. The semantics of this abstraction are such that different members of the group have consistent perceptions of the communication done in the group. The abstraction is typically implemented through the integration of two types of services: membership and reliable multicast.

Membership services maintain information about membership of groups. The membership of a group can change dynamically due to new processes joining and current members departing, failing, or disconnecting. The membership service tracks these changes and reports them to group members. The report given by the membership service to a member is called a *view*. It includes a unique identifier and a list of currently active and mutually connected members. Failures can partition a group into disconnected components of mutually connected members. Membership services strive to form and deliver the same views to all mutually connected members of the group.[1] While this is not always possible, they typically succeed once network connectivity more or less stabilizes (see, for example, [31, 16]).

In addition, GCSs provide reliable multicast services that allow application processes to send messages to the entire membership of a group. GCSs guarantee that message delivery satisfies certain properties. For example, one property can be that messages sent by the same sender are delivered in the order in which they were sent; another property can ensure that all processes receive all messages in the same total order. Different GCSs differ in the specific message delivery properties they provide,

[1]In this paper, we consider *partitionable* membership services, which may deliver concurrent, disjoint views of the same group to disconnected members.

but most of them provide some variant of *VS* semantics. We refer to a GCS providing such semantics as a *virtually synchronous GCS*, and to an algorithm implementing this semantics as a *VS algorithm*.

VS semantics specifies how message deliveries are synchronized with view deliveries. This synchronization is done in a way that simulates a "benign" world in which message delivery is reliable within each view. Many variants of VS have been suggested (for example, [38, 23, 16, 12, 40, 9]). Nearly all of them include a key property, called *virtually synchronous delivery*, which guarantees that *processes that receive the same pair of views from the GCS receive the same sets of messages in between receiving the views.* Henceforth, when we refer to VS, we assume the semantics includes virtually synchronous delivery.

EXAMPLE 1.1. *Assume Alice, Bob, and Carol are playing an online game. Assume they communicate using totally ordered messages and modify their game states when they receive messages. Each of them is initially given a view* $\langle\{Alice, Bob, Carol\}, 1\rangle$*, where* $\{Alice, Bob, Carol\}$ *is a set of members and* 1 *is a view identifier. Then Carol disconnects, and Alice and Bob are given a new view* $\langle\{Alice, Bob\}, 2\rangle$*. The virtually synchronous delivery property guarantees that both Alice and Bob receive the same messages before receiving the new view. In particular, if Bob receives a message from Carol before it receives the new view, then Alice also receives this message before the new view. Therefore, Alice and Bob remain in consistent states and can safely continue playing the game after they receive the new view.*

In general, virtually synchronous GCSs are especially useful for building applications that maintain a replicated state of some sort using a variant of the well-known *state-machine/active replication* approach [34, 41] and [32, Chapter 10]. With such an approach, processes that maintain state replicas are organized into multicast groups. Actions that update the state are sent using a multicast primitive that delivers messages to different processes in the same order. When processes receive these actions, they apply them to their local replicas. VS guarantees that processes that remain connected receive the same messages. This implies that processes that remain connected apply the same sequences of actions to their replicas. Hence, their replicas remain mutually consistent. Examples of GCS applications that use this technique are [2, 4, 29, 42, 24, 8].

Let us consider what is involved in implementing the virtually synchronous delivery property. Imagine that GCS processes are forming a new view because someone has disconnected from their current view. The GCS processes must make sure that they deliver the same messages to their application clients before delivering to them the new view. However, it may be the case that some of these GCS processes received messages that others did not. In the scenario illustrated in Example 1.1, the last messages from Carol may have reached the GCS process of only Bob, and not of Alice; Bob and Alice need to agree on whether or not to deliver these messages. To ensure such agreement, GCS processes invoke a *synchronization* protocol whenever a new view is forming.

Designing correct and efficient algorithms that implement VS is not trivial. Different GCS processes may perceive connectivity changes inconsistently. Since the desired synchronization depends on who the members of the new view are, the algorithm has to tolerate transient inconsistent views and cascading connectivity changes.

In particular, a VS algorithm needs to know which synchronization messages sent by different processes pertain to the same view formation attempt. Existing algorithms, such as [23, 3, 40, 9, 26, 5], identify such synchronization messages by tagging them with a common identifier. Some initial communication is performed first, before synchronization messages are communicated, in order to agree upon a common identifier and to distribute it to the members of the forming view.

While a view is forming and a synchronization protocol is executing, there may be changes in connectivity that call for views with altogether different memberships. When such situations happen, existing VS algorithms, for example [23, 26, 40, 9, 5], continue executing their current synchronization protocol to termination and then deliver to the application a view that does not reflect the already detected changes in connectivity; we refer to such views as *obsolete* [31]. Afterwards, the algorithm is invoked anew to incorporate the new changes. Obsolete views cause an overhead not just for the GCS but also for applications. Since application processes do not know when the views delivered to them are obsolete, they handle such views just as they do any other view, for example, by running state synchronization protocols [29, 22, 33].

1.2. Our contributions. In this paper, we present a novel design for a virtually synchronous GCS targeted for WANs. We make the following contributions:

1. We present a new algorithm for implementing VS. Our algorithm neither processes nor delivers views with obsolete memberships. Moreover, the synchronization protocol run by our algorithm involves just a single message exchange round among members of the new view. We are not aware of any other algorithm for implementing VS that has these two features.

2. Our design demonstrates how to effectively decouple the algorithm for achieving VS from the algorithm for maintaining membership. As suggested in [6, 31], such effective decoupling is important for providing scalable GCSs in WANs.

We define a membership service interface that allows the VS algorithm to execute in parallel with the membership algorithm. In contrast to previous designs, for example, [40, 11], we allow the membership algorithm to freely change memberships of forming views at any time. Moreover, the interaction between the membership and VS algorithms is only in one direction, from the former to the latter. Our interface was adopted by the Moshe [31] membership algorithm; other existing membership algorithms (for example, [20, 5]) can also be easily extended to provide the required interface and semantics.

3. Our design is carried out much more rigorously and formally than most previous designs of virtually synchronous GCSs. The presented specifications of our GCS and its environment, description of the algorithm, and proof of correctness are all precise and formal. Our project is the first to use formal methods for modeling a virtually synchronous GCS and to provide an assertional proof of its correctness.

Our algorithm has been implemented [43] (in C++) as part of a novel architecture for scalable group communication in WANs using the datagram service of [7] and the Moshe membership algorithm [31].

1.3. Roadmap. The rest of this paper is organized as follows. Section 2 gives an overview of our algorithm and overall design. Section 3 reviews the formal model and notation. In section 4 we present the client-server architecture of our GCS and formally specify the assumptions we make on the membership service and the underlying communication substrate. Section 5 contains precise specifications of the safety and liveness properties satisfied by our GCS. The algorithm is then given in section 6 and is accompanied by informal correctness arguments. Section 7 concludes the paper. A formal correctness proof that the algorithm of section 6 satisfies the specifications of section 5 is given in the appendixes: safety properties are given in Appendix B, and liveness properties are given in Appendix C. Appendix A reviews the proof techniques used in Appendixes B and C.

2. Design overview. The novelty of our algorithm for achieving VS is concentrated in its synchronization protocol. Recall that this protocol is run among GCS processes in order for those that remain connected to agree upon a common set of messages each of them must deliver before moving into the new view. The protocol depends on a simple yet powerful idea. Instead of using common identifiers to designate which synchronization messages pertain to the same view formation attempt, we use locally generated identifiers. These identifiers are then included as part of the formed views.[2] Once a view formation completes at a GCS process, the process knows which synchronization messages of other members to consider for the view—the messages tagged with the identifiers that are included in the view.

EXAMPLE 2.1. *View* $\langle 8, \{Alice, Bob, Carol\}, [4,3,7]\} \rangle$ *has membership* $\{Alice, Bob, Carol\}$*, vector of local identifiers* $[4,3,7]$*, and view identifier* 8*. When a GCS process forms this view, it uses the synchronization messages from Alice, Bob, and Carol tagged, respectively, with* 4*,* 3*, and* 7 *to decide on the set of messages it must deliver before delivering this view to its application. Thus, if Alice, Bob, and Carol form the same view, they use the same synchronization messages, and thus agree on which application messages each of them needs to deliver.*

The use of local identifiers eliminates the need to preagree on common identifiers and allows the synchronization protocol to complete in a single message exchange round. It also allows the algorithm to promptly react to connectivity changes without wasting resources on obsolete views. The protocol works correctly even if, because of network instability, GCS processes send multiple synchronization messages during the same synchronization protocol.

2.1. Architecture for WAN. Our design decouples the algorithm for implementing virtually synchronous multicast from the algorithm for maintaining membership. The membership algorithm handles generation of local identifiers and formation of views. The algorithm for implementing virtually synchronous multicast synchronizes views and application messages to implement the VS semantics. In particular, it handles multicast requests submitted by the application, delivers application messages and views back to the application, and runs the synchronization protocol to synchronize processes that transition together into new views. The decoupling involves low-cost, one-directional communication from the membership to the virtually synchronous multicast algorithm. It also allows the synchronization protocol to execute in parallel with the membership algorithm forming views.

Efficient decoupling of membership and virtually synchronous multicast algorithms allows for an architecture in which the membership service is implemented by a small set of dedicated membership servers maintaining the membership information on behalf of a large set of clients. This architecture was proposed in [6, 31] for supporting scalable membership services in WANs. Our work extends this architecture by specifying how it can be used as a base for a virtually synchronous GCS. In particular, we present precise specifications of the interface and semantics that a membership service has to provide in order to be decoupled from the virtually synchronous multicast algorithm.

The interface consists of two types of messages, `start` and `view`, sent from membership servers to the processes executing the virtually synchronous multicast algo-

[2] A similar view structure is suggested in [40] for the purpose of not having concurrent views intersect.

rithm; we call these processes the GCS end-points. A `start` message is sent when a membership server starts forming a new view or adds new members to an already forming view. Each `start` message contains a prospective membership set and an identifier, which is *not* globally agreed upon; that is, different processes can be given different identifiers. A `view` message is sent when a server succeeds in forming a new view. The `view` message contains information that maps GCS end-points to the last start identifiers they were given prior to this view. The servers do not need to hear back from the end-points in order to complete the membership algorithm, and the end-points do not impose any restrictions on the servers' choices of views. These two features are in contrast with previously suggested external membership services, such as, for example, Maestro [11] and the service of [40], in which membership servers are not allowed to add new members once view formation begins and, furthermore, have to synchronize with the processes executing the virtually synchronous multicast algorithm before they can produce new views.

2.2. Algorithm for virtual synchrony. When the membership service starts forming a view, it sends a `start` notification with a prospective membership set and a new local identifier to each end-point `p` executing the virtually synchronous multicast algorithm. Upon receiving this notification, `p` sends a synchronization message tagged with this identifier to the end-points in the prospective membership set. In the synchronization message, `p` specifies its current view and the set of application messages that `p` commits to deliver in its current view before delivering the new view. If an end-point `q` joins the membership while a view formation is in progress, `p` will receive a new `start` notification and will then forward to `q` the same synchronization messages it sent when the view formation started.

When `p` receives a `view` message from its membership server, the local identifiers included in the view tell `p` which synchronization messages to consider—those messages that are tagged with the local identifiers included in the new view. Using the information contained in these messages, `p` computes two things: (a) the set of end-points, called the *transitional set* [16], containing those members of the new view that would transition into the new view together with `p` directly from `p`'s current view; and (b) the set of application messages that `p` must deliver in its current view before transitioning into the new view. End-point `p` computes the transitional set to include every end-point `q` that is a member of both `p`'s current view and the new view, and whose synchronization message was sent in the same view as `p`'s current view. As far as the set of application messages, end-point `p` decides on delivering the maximal set of messages identified by the synchronization messages of the transitional set members. Since the same views formed by different end-points contain the same local identifiers, the end-points use the same synchronization messages to compute the transitional set and the message set. After delivering the decided set of application messages, `p` delivers the new view and the transitional set to its application client. The transitional set tells the client about the members of the new view with whom the client is already synchronized.

Unlike previous algorithms, for example, those in [5, 26, 9, 40], our algorithm allows the membership service to change the membership of a forming view while the synchronization protocol is running; the protocol responds immediately to such membership changes. The following example demonstrates the benefits of this approach.

EXAMPLE 2.2. *Figure* 2.1 *presents a sample execution involving two clients,* `a` *and* `b`*. Vertical arrows represent the time passage at each client, empty circles represent* GCS*-level events, and gray circles represent* MEMB*-level events. In order*

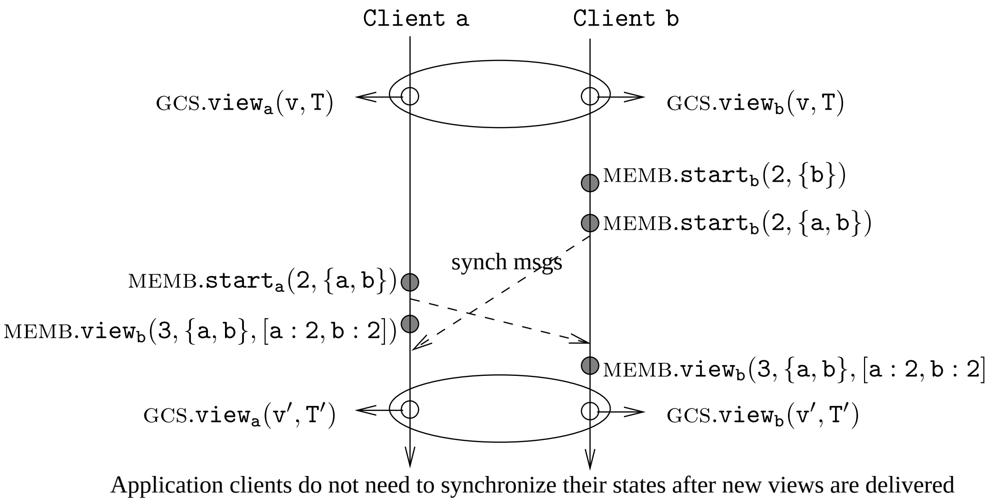

FIG. 2.1. *Handling membership changes while synchronization protocol is running.*

to disambiguate these events, we prefix events generated at the membership server by MEMB *and events generated by the virtually synchronous multicast algorithm by* GCS.

First, both clients receive the same view $\mathtt{v} = \langle 2, \{\mathtt{a}, \mathtt{b}\}, [\mathtt{a} : 1, \mathtt{b} : 1]\rangle$ *from their* GCS *end-points,* $\text{GCS}_\mathtt{a}$ *and* $\text{GCS}_\mathtt{b}$*; the ellipse around these view events highlights that the delivered views are the same. At some point, the* MEMB *service notifies* $\text{GCS}_\mathtt{b}$ *that it is starting to form a view without* a*. While doing so, it detects that* a *is connected to* b *after all, so it changes the membership of the forming view to* $\{\mathtt{a}, \mathtt{b}\}$*.* $\text{GCS}_\mathtt{b}$ *forwards to* $\text{GCS}_\mathtt{a}$ *its latest synchronization message; synchronization messages are denoted by dashed lines.* $\text{GCS}_\mathtt{a}$ *is also notified by* MEMB *of its attempt to form a new view with* b*; this causes* $\text{GCS}_\mathtt{a}$ *to send a synchronization message to* $\text{GCS}_\mathtt{b}$*. When* MEMB *completes its view formation, it delivers the new view* $\mathtt{v}' = \langle 3, \{\mathtt{a}, \mathtt{b}\}, [\mathtt{a} : 2, \mathtt{b} : 2]\rangle$ *to both* GCS *end-points. After the* GCS *end-points receive each others' synchronization messages, they compute their transitional sets to be* $\mathtt{T}' = \{\mathtt{a}, \mathtt{b}\}$*, decide on which application messages they need to deliver, deliver these messages, and then deliver* $\mathtt{v}'$ *and* $\mathtt{T}'$ *to their clients. From* $\mathtt{T}'$*,* a *and* b *can deduce that, due to virtually synchronous delivery, they received the same messages while in* v*, and therefore do not need to synchronize their states.*

Example 2.2 demonstrates two additional advantages of our algorithm: (a) the algorithm does not waste resources on synchronizing end-points in order to deliver views that are known to be obsolete; and (b) the application benefits from not seeing obsolete views, as it has to do fewer state synchronizations (or other view processing activity). Responding promptly to connectivity changes is therefore especially important in WANs, where transient connectivity changes may occur frequently due to variability of message latency and less reliable connectivity. In contrast to our algorithm, algorithms that do not allow new members to be added to the membership of an already forming view (such as [5, 26, 9, 40]) lack these advantages, as illustrated by the following example.

EXAMPLE 2.3. *When executed in the scenario of Example* 2.2*, algorithms that do not allow new members to be added to the membership of an already forming view would deliver an obsolete view* $\mathtt{v_{mid}}$ *with membership* $\{\mathtt{b}\}$ *to client* b *and then restart* *the view formation and synchronization protocols anew in order to deliver to* a *and* b *a new view with membership* $\{a, b\}$*. As part of the synchronization protocol,* a *and* b *would first exchange messages to agree upon a common identifier before actually exchanging synchronization messages. The synchronization protocol would* not *synchronize end-points* a *and* b *because they would be transitioning into the new view from different views,* a *from* v *and* b *from* v_{mid}*. As a result, after the clients get the new view from* GCS*, they would have to run an additional state synchronization protocol.*

2.3. Formal methodology. Our design has been carried out and is presented at a level more formal and rigorous than that of most previous designs of virtually synchronous GCSs. We precisely specify the properties satisfied by our virtually synchronous multicast algorithm, the external membership service, and the underlying communication substrate. We then give a formal description of the virtually synchronous multicast algorithm. The algorithm is accompanied by a careful formal correctness proof. The safety properties are proved by using invariant assertions and simulation mappings; the liveness properties are proved by using invariant assertions and careful operational arguments. We found this level of rigor to be important: in the process of specifying and verifying the algorithm, we uncovered several ambiguities and errors.

Previously, formal approaches were used to specify the semantics of virtually synchronous GCSs and to model and verify their applications, for example, in [15, 22, 18, 33, 27]. Existing algorithms implementing VS are modeled in pseudocode and proven correct operationally. However, due to their size and complexity, such algorithms were not previously modeled using formal methods nor were they assertionally verified.

To manage the complexity of this project we have developed a formal inheritance-based methodology [30] for incrementally constructing specifications, algorithms, and proofs. In addition to making the project tractable, the use of this construct makes clear which parts of the algorithm implement which property. The modularity of this approach facilitates further modifications and alterations of the design. Our project and the inheritance-based construct are both developed in the framework of the I/O automaton formalism (see [37] and [36], Chap. 8).

3. Formal model and notation. In the I/O automaton model ([37] and [36], Chap. 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.

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 contains either 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 solely consisting of the automaton's external actions. A *fair trace* is a trace of a fair execution.

When reasoning about an automaton, we are interested in only 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 centralized, global, I/O automata that generate the legal sets of traces; for such automata we do not specify task partitions. Each external action in such a centralized automaton is tagged with a subscript which denotes the process at which this action occurs. An algorithm automaton *satisfies* a specification if all of its traces are also traces of the specification automaton. Refinement mappings are a 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 proof techniques are reviewed in Appendix A. Liveness properties usually specify that some good thing eventually happens. An algorithm 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. 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 prestate is modified *atomically* to yield the poststate. The precondition-effect style is also known as a *guarded command* style: events have guards, or preconditions, and are triggered when the preconditions are enabled.

We have developed a novel formal notion of inheritance for automata [30]. A *child* automaton is specified as a modification of the parent automaton's code. When presenting a child we first specify a *signature extension* which consists of new actions, labeled new, and modified actions. A modified action is labeled with the name of the action which it modifies 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 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 [30].

Inheritance allows us to reuse code and avoid redundancies. It also allows us to reuse proofs: Assume that an algorithm automaton `A` can simulate a specification automaton `S`, and let $\mathtt{A}'$ and $\mathtt{S}'$ be child automata of `A` and `S`, respectively. Then the proof extension theorem of [30] asserts that in order to prove that $\mathtt{A}'$ can simulate $\mathtt{S}'$, it is sufficient to show that the restrictions added by $\mathtt{A}'$ are consistent with the restrictions $\mathtt{S}'$ places on `S` and that the new functionality of $\mathtt{A}'$ can simulate new

FIG. 4.1. *The client-server architecture: GCS end-points using an external membership service. Arrows represent interaction between GCS end-points and underlying services.*

functionality of $\mathtt{S}'$. Appendix A contains more details.

4. Client-server architecture and environment specification. Our service is designed to operate in an asynchronous message-passing environment. Processes and communication links may fail and may later recover, possibly causing network partitions and merges. For simplicity, we assume that processes recover with their running state intact; this is a plausible assumption as processes can keep their running state on stable storage. We do not explicitly model process crashes and recoveries because under this assumption a crashed process is indistinguishable from a slow one. In section 6.4, we argue that our algorithm also provides meaningful semantics when group communication processes lose their entire state upon a crash and recover with their state reset to an initial value.

Our GCS is implemented by a collection of GCS *end-points*, which are the GCS processes that run at the application clients' locations. GCS end-points handle clients' multicast requests and inform their clients of view changes.

The GCS architecture is depicted in Figure 4.1. All GCS end-points run the same algorithm. The algorithm relies on the underlying membership and multicast services to handle, respectively, formation of views and transmission of messages. The algorithm's task is to synchronize output of the two underlying services to implement the VS semantics.

Sections 4.1 and 4.2 below give precise specifications of the interface and semantics that the underlying membership and multicast services have to provide in order to be suitable for our algorithm. Services that satisfy these (or very similar) requirements have been previously used for GCSs, and efficient implementations of these services for WANs exist; see, for example, [31, 7].

4.1. The membership service specification. This section presents a formal specification of the membership services that are appropriate for our GCS design. For simplicity, here and in the rest of the paper, we assume that there is a single process group; multiple groups can be supported by treating each independently. We also omit part of the interface that handles processes' requests to join and leave groups.

Figure 4.2 contains an I/O automaton, called MEMB, that defines the interface and the safety properties of the membership service. The service interface is given by the automaton's signature.[3] Informally, it consists of the following two output

[3]When specifying a distributed system as a centralized automaton, we subscript each external action of the specification automaton with the location (or process) in the distributed system at which the action occurs.

```
AUTOMATON MEMB

Type:
    Proc: Set of end-points.
    StartId: Total-order; cid_0 is smallest.
    ViewId: Partial-order; vid_0 is smallest.
    View: ViewId × SetOf(Proc) × (Proc → StartId).
Def:  v_p = ⟨vid_0, {p}, {(p →cid_0)}⟩.

Signature:
 Output: start_p(cid, set), Proc p, StartId cid, SetOf(Proc) set
         view_p(v), Proc p, View v

State:
 For all Proc p: View  memb_view[p], initially v_p
 For all Proc p: (StartId × SetOf(Proc)) start[p], initially ⟨cid_0, {}⟩

Transitions:
 OUTPUT  start_p(cid, set)
 pre: cid > memb_view[p].startId(p)
      cid ≥ start[p].id
      p ∈ set
 eff: start[p] ← ⟨cid, set⟩

 OUTPUT  view_p(v)
 pre: p ∈ v.set ∧ v.id > memb_view[p].id
      v.set ⊆ start[p].set
      v.startId(p) = start[p].id
      v.startId(p) > memb_view[p].startId(p)
 eff: memb_view[p] ← v
```

FIG. 4.2. *Membership service interface and safety specification.*

actions:

$\mathtt{start_p(cid, set)}$ notifies process p that the membership service is attempting to form a view with the members of set; cid is a local start identifier;

$\mathtt{view_p(v)}$ notifies process p that the membership service has succeeded in forming view v. A view v is a triple consisting of an identifier v.id, a set of members v.set, and a function v.startId that maps members of v to start identifiers. Two views are the same if they consist of identical triples.

Automaton MEMB maintains two state variables, memb_view[p] and start[p], for each client p. These variables contain, respectively, the last view and the last start message issued to client p; the variables are updated in the effects of the transitions. The safety properties satisfied by the MEMB automaton include two basic properties, which are provided by virtually all group membership services (for example, [13, 20, 5, 23, 9, 31, 40, 3]), as well as some new properties concerning the start notifications.

The two basic properties are *self-inclusion* and *local monotonicity*. Self-inclusion requires every view issued to a client p to include p as a member; this property is enforced with a precondition $\mathtt{p} \in \mathtt{v.set}$ on the $\mathtt{view_p(v)}$ action. Local monotonicity requires that view identifiers delivered to p be monotonically increasing; this property is enforced with a precondition $\mathtt{v.id} > \mathtt{memb\_view[p]}$ on the $\mathtt{view_p(v)}$ action. Local monotonicity has two important consequences: the same view is not delivered more than once to the same client, and clients that receive the same two views receive them in the same order [16].

In addition, the MEMB automaton specifies that the membership service must issue at least one start notification to client p before issuing a new view v to p. Also, the start identifier v.startId(p) contained in the new view v must be the same as the identifier of the latest preceding start issued to p. These two requirements are enforced by the last two preconditions on $\mathtt{view_p(v)}$. In particular, the former one is achieved by requiring that a bigger start identifier than the one associated with p in the last view has been issued to p.

The MEMB specification allows the membership service to react to connectivity changes happening during view formation. Whenever the service wants to add new

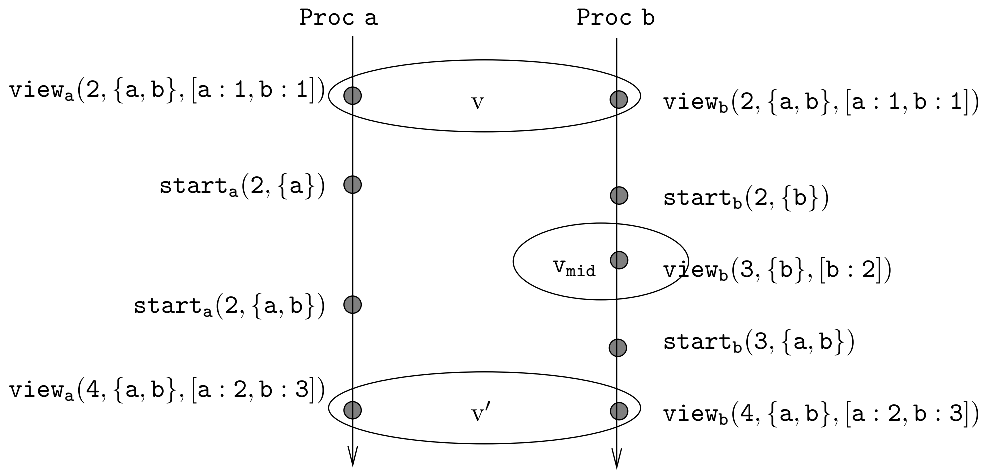

FIG. 4.3. *A sample execution of* MEMB.

members to the membership, it has to issue a new `start` notification to the clients: the second precondition on $\mathtt{view}_\mathtt{p}(\mathtt{v})$ actions requires the membership `v.set` to be a subset of the tentative membership `set` included in the last `start` notification. In order to remove members from a forming view, the service does not need to issue a new `start` notification.

The first `start` notification issued to `p` after a view marks the beginning of a new view formation period. It includes a new local identifier `cid`, different from the ones that were previously sent to `p`: the first precondition on $\mathtt{start}_\mathtt{p}(\mathtt{cid}, \mathtt{set})$ requires `cid` to be strictly greater than memb_view[p].startId(p). Subsequent `start` notifications sent during an ongoing view formation may either reuse the last start identifier or issue a new one, as specified by the second precondition on `start` actions. We ensure uniqueness of local start identifiers by generating them in increasing order.

Notice that the MEMB automaton does not specify any relationship between views issued to different clients.

EXAMPLE 4.1. *Figure* 4.3 *presents a sample execution that shows the* MEMB *service delivering different sequences of views to two different clients,* a *and* b*. Arrows represent time passage at each client; gray dots represent events. First, both clients receive the same view* $\mathtt{v} = \langle 2, \{\mathtt{a}, \mathtt{b}\}, [\mathtt{a} : 1, \mathtt{b} : 1]\rangle$*; we illustrate this with a circle around the view events at both clients. Then, client* b *receives a view* $\mathtt{v}_{\mathtt{mid}} = \langle 3, \{\mathtt{b}\}, [\mathtt{b} : 2]\rangle$ *by itself. Then, both clients receive another common view* $\mathtt{v}' = \langle 4, \{\mathtt{a}, \mathtt{b}\}, [\mathtt{a} : 2, \mathtt{b} : 3]\rangle$*. Notice how the start identifiers included in the views correspond to the last start identifiers issued to the clients.*

We do *not* specify liveness properties for membership services. Instead, when we specify the liveness properties of our GCS in section 5.2, we *condition* them on the behavior of the membership service. For example, we state that if the same view is delivered to all the members and the members do not receive any subsequent membership events, then they eventually deliver this view to their application clients. Existing membership services do satisfy meaningful liveness properties. For example, [31] guarantees that, when the network stabilizes, all members receive the "correct" view and no other views thereafter. By combining our GCS liveness properties with such membership liveness properties, we can restate the liveness properties of our GCS

```
AUTOMATON CO_RFIFO

Signature:
Input:                                              Output:   deliver_p,q(m), Proc p, Proc q, Msg m
  send_p(set,m), Proc p, SetOf(Proc) set, Msg m     Internal: lose(p,q), Proc p, Proc q
  reliable_p(set), Proc p, SetOf(Proc) set                    skip_task(p,q), Proc p, Proc q
  live_p(set), Proc p, SetOf(Proc) set
State:
 For all Proc p, Proc q: SequenceOf(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)                               INTERNAL lose(p, q)
 eff: (∀ q ∈ set) append m to channel[p][q]         pre: q ∉ reliable_set[p]
                                                    eff: dequeue last message from channel[p][q]
 OUTPUT deliver_p,q(m)
 pre: m = first(channel[p][q])                      INPUT live_p(set)
 eff: dequeue m from channel[p][q]                  eff: live_set[p] ← set

 INPUT reliable_p(set)                              INTERNAL skip_task(p, q)
 eff: reliable_set[p] ← set                         pre: q ∉ live_set[p]
Tasks:
For each Proc p, Proq q: C_p,q = ({deliver_p,q(m) | m ∈ Msg} ∪ {skip_task(p,q)} ∪ {lose(p,q)})
```

FIG. 4.4. *Reliable* FIFO *multicast service specification. Liveness-related code is italicized.*

conditionally on the network behavior.

The MEMB specification allows for simple and efficient distributed implementations that also satisfy meaningful liveness properties. The membership service of [31] is an example of such an implementation; our design was implemented by Tarashchanskiy [43] using this membership service. In this service, a small number of servers support a large number of clients, communicating with them asynchronously via FIFO ordered channels (TCP sockets). In case a server fails, clients can migrate to another server. Other existing membership algorithms (for example, [20, 5]) could also be easily extended to provide the interface and semantics specified here.

4.2. The reliable FIFO multicast service specification. The group communication end-points communicate with each other using an underlying multicast service that provides reliable FIFO communication between every pair of connected processes. Many existing group communication systems (for example, [26, 9, 20, 3]) implement VS over similar communication substrates. In our implementation [43], we use the service of [7].

Figure 4.4 presents an I/O automaton, CO_RFIFO, that specifies a multicast service appropriate for our GCS design. Portions of the code that define liveness properties are italicized.

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

In addition, the interface of CO_RFIFO includes input actions of the type $\mathtt{reliable}_{\mathtt{p}}(\mathtt{set})$; end-point p may use such actions to command the multicast service to maintain a reliable (gap-free) FIFO connection to the end-points listed in set. Whenever this action occurs, set is stored in a special variable reliable_set[p]. For every process q not in reliable_set[p], the multicast service may lose an arbitrary suffix of the messages sent from p to q, as modeled by an internal action lose(p, q).

In order for the multicast service to be considered live, messages sent to live and connected processes must eventually reach their destinations. The CO_RFIFO specification enforces this property in the italicized portion of its code.

Recall from section 3 that an infinite fair execution of an automaton must contain either infinitely many events from each task $\mathtt{C}$ or infinitely many occurrences of states in which no action in $\mathtt{C}$ is enabled. Automaton CO_RFIFO defines the set $\mathtt{C}_{\mathtt{p,q}} = (\{\mathtt{deliver}_{\mathtt{p,q}} \mid \mathtt{m} \in \mathtt{Msg}\} \cup \mathtt{skip\_task}(\mathtt{p},\mathtt{q}) \cup \mathtt{lose}(\mathtt{p},\mathtt{q}))$ to be a task for each pair of end-points $\mathtt{p}$ and $\mathtt{q}$. This definition implies that $\mathtt{deliver}_{\mathtt{p,q}}$ actions must occur in an infinite fair execution of CO_RFIFO, provided the following three conditions hold: there are messages sent from $\mathtt{p}$ to $\mathtt{q}$—hence, $\mathtt{deliver}_{\mathtt{p,q}}$ is enabled; the client at $\mathtt{p}$ is interested in maintaining reliable connection to $\mathtt{q}$—hence, $\mathtt{lose}(\mathtt{p},\mathtt{q})$ is disabled; and $\mathtt{q}$ is believed to have a live connection to $\mathtt{p}$—hence, a special action $\mathtt{skip\_task}(\mathtt{p},\mathtt{q})$ is disabled, as explained below.

Action $\mathtt{skip\_task}(\mathtt{p},\mathtt{q})$ is defined only to provide an alternative to $\mathtt{deliver}_{\mathtt{p,q}}$ actions so that $\mathtt{deliver}_{\mathtt{p,q}}$ actions are not required to happen when $\mathtt{q}$ is believed to be disconnected from $\mathtt{p}$. $\mathtt{skip\_task}(\mathtt{p},\mathtt{q})$ is an internal action that has no effect on the state of CO_RFIFO and is enabled when $\mathtt{q}$ is believed to be disconnected from $\mathtt{p}$. Such belief is modeled using special $\mathtt{live}_{\mathtt{p}}(\mathtt{set})$ input actions. The $\mathtt{set}$ argument is assumed to represent a set of processes that are alive and connected to $\mathtt{p}$; when such an input happens, $\mathtt{set}$ is stored in a state variable $\mathtt{live\_set}[\mathtt{p}]$. The precondition on the $\mathtt{skip\_task}(\mathtt{p},\mathtt{q})$ action is $\mathtt{q} \notin \mathtt{live\_set}[\mathtt{p}]$.

An important implication of how tasks are defined in CO_RFIFO is that, if $\mathtt{q}$ remains in both $\mathtt{live\_set}[\mathtt{p}]$ and $\mathtt{reliable\_set}[\mathtt{p}]$ from some point on in a fair execution of CO_RFIFO, then all the messages that $\mathtt{p}$ sends to $\mathtt{q}$ from that point on are eventually delivered to $\mathtt{q}$.

5. Specifications of the group communication service. The next two subsections contain specifications of the safety and liveness properties satisfied by our GCS. The specifications capture a core set of properties that is commonly provided by GCSs and that have been shown to be useful for facilitating implementations of many distributed applications and other, stronger, group communication properties (see [16]). For example, [32, Chapter 10] illustrates the utility of our GCS system by describing a simple application that can be effectively built using GCS. The application implements a variant of a data service that allows a dynamic group of clients to access and modify a replicated data object.

5.1. Safety properties. We present the safety specification of our GCS incrementally as four automata: In section 5.1.1 we specify a simple GCS that synchronizes delivery of views and application messages to require *within-view delivery* of messages. In section 5.1.2 we extend the specification of section 5.1.1 to also require *virtually synchronous delivery*, the key property of VS (see section 1.1). In section 5.1.3 we specify the *transitional set* property, which complements virtually synchronous delivery. Finally, in section 5.1.4, we specify the *self-delivery* property, which requires the GCS to deliver to each client the client's own messages.

The incremental development of the safety specification is matched later when we develop the algorithm and its correctness proof in section 6 and Appendix B.

5.1.1. Within-view reliable FIFO multicast. In this section we specify a GCS that captures the following properties:

1. Views delivered to the application satisfy the self-inclusion and local monotonicity properties of the MEMB service; see section 4.1.

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 all Proc p, View v: SequenceOf(AppMsg) msgs[p][v], initially empty
 For all Proc p, Proc q: Int last_dlvrd[p][q], initially 0
 For all 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
```

Fig. 5.1. WV_RFIFO *service specification.*

2. Messages are delivered in the same view in which they were sent. This property is useful for many applications (see [23, 16, 42]) and appears in several systems and specifications (for example, [13, 44, 5, 38, 22, 28, 18]). A weaker property that requires each message to be delivered in the same view at every process that delivers it, but not necessarily the view in which it was sent, is typically implemented on top of an implementation of within-view delivery (see [16]).

3. Messages are delivered in gap-free FIFO order (within views). This is a basic property upon which one can build services with stronger ordering guarantees, such as causal order or total order. The totally ordered multicast algorithm of [14] is implemented atop a service with a similar specification.

Figure 5.1 presents automaton WV_RFIFO : SPEC that models this specification. The automaton uses centralized queues msgs[p][v] of application messages for each sender p and view v. It also maintains a variable current_view[p] that contains the last view delivered to each process p and a variable last_dlvrd[q][p], for every pair of processes q and p, containing the index in the msgs[q][current_view[p]] queue of the last message from q delivered to p in p's current view.

Action $\texttt{view}_\texttt{p}(\texttt{v})$ models the delivery of view v to process p; the precondition on this action enforces self-inclusion and local monotonicity. Action $\texttt{send}_\texttt{p}(\texttt{m})$ 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]]. Action $\texttt{deliver}_\texttt{p}(\texttt{q}, \texttt{m})$ models the delivery to process p of message m sent by process q. The gap-free FIFO ordered delivery of messages within-views is enforced by its precondition, which allows delivery of only the message indexed by last_dlvrd[q][p] + 1 in the msgs[q][current_view[p]] queue.

5.1.2. Virtually synchronous delivery. In this section we use the inheritance-based methodology to modify the WV_RFIFO : SPEC automaton to also enforce the *virtually synchronous delivery* property. The modified automaton, VSRFIFO : SPEC, is defined by the code contained in both Figures 5.1 and 5.2.

Figure 5.2 contains the code that enforces the *virtually synchronous delivery* property. Recall from section 1.1 that this property requires processes moving together from view v to view v′ to deliver same set of messages while in view v. Since the parent specification, WV_RFIFO : SPEC, imposes gap-free FIFO delivery of messages, a message set can be represented by a set of indices, each pointing to the last message from each member of v; such representation of a set is called a *cut*.

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 all 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
```

FIG. 5.2. VS_RFIFO *service specification.*

The WV_RFIFO : SPEC automaton fixes a cut for processes that wish to move from some view $\mathtt{v}$ to some view $\mathtt{v}'$: A new internal action $\mathtt{set\_cut}(\mathtt{v},\mathtt{v}',\mathtt{c})$ sets a new variable $\mathtt{cut}[\mathtt{v}][\mathtt{v}']$ to a cut mapping $\mathtt{c}$. For a given pair of views, $\mathtt{v}$ and $\mathtt{v}'$, the cut is chosen only once, *nondeterministically.* Delivery of a view $\mathtt{v}$ to process $\mathtt{p}$ is allowed only if a cut for moving from $\mathtt{p}$'s current view into $\mathtt{v}$ has been set and if $\mathtt{p}$ has delivered all the messages identified in this cut. These conditions are enforced by the two new preconditions of the $\mathtt{view_p}(\mathtt{v})$ action (see Figure 5.2). Since VSRFIFO : SPEC is a modification of WV_RFIFO : SPEC, the new preconditions work in conjunction with the preconditions in $\mathtt{view_p}(\mathtt{v})$ of WV_RFIFO : SPEC.

The VSRFIFO : SPEC automaton, being a safety specification, does not require liveness properties to hold, for instance, that processes actually deliver messages specified by the cuts and hence are able to satisfy conditions for delivering new views. Such liveness specifications are stated in section 5.2.

5.1.3. Transitional set. While virtually synchronous delivery is a useful property, a process that moves from view $\mathtt{v}$ to view $\mathtt{v}'$ cannot tell locally which of the processes in $\mathtt{v.set} \cap \mathtt{v}'\mathtt{.set}$ move to view $\mathtt{v}'$ directly from view $\mathtt{v}$ and which move to $\mathtt{v}'$ from some other view. In order for the application to be able to exploit the virtually synchronous delivery property, application processes need to be informed which other processes move together with them from their current view into their new view. The set of processes that transition together from one view into the next is called a *transitional set* [16].

DEFINITION 5.1. *A transitional set from view* $\mathtt{v}$ *to view* $\mathtt{v}'$ *is a subset of* $\mathtt{v.set}\cap \mathtt{v}'\mathtt{.set}$ *that includes* (a) *all processes that receive view* $\mathtt{v}'$ *while in view* $\mathtt{v}$ *and* (b) *no process that receive view* $\mathtt{v}'$ *while in a view other than* $\mathtt{v}$.

Note that the transitional set is not uniquely defined by Definition 5.1. If a process $\mathtt{p}$ in $\mathtt{v.set} \cap \mathtt{v}'\mathtt{.set}$ does not receive view $\mathtt{v}'$, Definition 5.1 does not specify whether or not $\mathtt{p}$ is included in the transitional sets of other processes that do receive view $\mathtt{v}'$.

The notion of a transitional set was first introduced as part of a special transitional view in the EVS [38] model. In our formulation (as in [16]), transitional sets are delivered to the application along with views as an additional parameter $\mathtt{T}$.

EXAMPLE 5.1. *Assume that Alice and Bob are using a virtually synchronous GCS that eventually reports the views produced by the* MEMB *service to Alice and Bob. Consider the scenario described in Example* 4.1*: both Alice and Bob receive views* $\mathtt{v}$ *and* $\mathtt{v}'$ *with the membership* {*Alice, Bob*}*. Just from these views, Alice does not know whether Bob receives view* $\mathtt{v}'$ *while in view* $\mathtt{v}$ *or while in some other view* $\mathtt{v_{mid}}$ *with the membership* {*Bob*}*. If the former holds, then Alice does not need to synchronize with Bob because virtually synchronous delivery guarantees that they have received the same messages while in view* $\mathtt{v}$*; otherwise, she does. The transitional set*

```
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 all Proc p:  View current_view[p], initially v_p
For all 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]
```

Fig. 5.3. *Transitional set specification.*

```
AUTOMATON WV_RFIFO+SELF : SPEC     MODIFIES  WV_RFIFO : SPEC

Signature Extension:
 Output: view_p(v) modifies wv_rfifo.view_p(v)

Transition Restriction:
 OUTPUT  view_p(v)
 pre: last_dlvrd[p][p] = LastIndexOf(msgs[p][current_view[p]])
```

Fig. 5.4. WV_RFIFO+SELF *service specification.*

given to Alice together with view v' *provides this information.*

Figure 5.3 presents an automaton TS : SPEC that specifies delivery of transitional sets (Definition 5.1). There are two types of actions: output actions $\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$ deliver view `v` and transitional set `T` to process `p`; and internal actions $\texttt{set\_prev\_view}_\texttt{p}(\texttt{v})$ declare that `q` intends to deliver view `v` while in its current view. The intentions are recorded in the variable `prev_view[p][v]`, and the current views are recorded in the variable `current_view[p]`.

Before process `p` can deliver a view `v`, each member `q` in the intersection of these views must execute $\texttt{set\_prev\_view}_\texttt{q}(\texttt{v})$, as enforced by the second precondition. The transitional set `T` delivered by `p` with `v` is then computed to consist of those processes `q` in the intersection `current_view[p].set` ∩ `v.set` for which `prev_view[q][v]` is the same as `current_view[p]`; this is specified by the third precondition on $\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$.

5.1.4. Self-delivery. We now specify the *self-delivery* property, which requires that each client receives all the messages it sent in a given view before receiving a new view. We specify this property as a simple modification of the WV_RFIFO : SPEC automaton presented in section 5.1.1; the modified automaton is defined by the code contained in both Figures 5.1 and 5.4.

In order to enforce self-delivery, a new precondition on the $\texttt{view}_\texttt{p}(\texttt{v})$ action requires the `last_dlvrd[p][p]` index to point to the last message sent by client `p` in its current view. Since the parent automaton, WV_RFIFO : SPEC, guarantees within-view gap-free FIFO delivery, this precondition implies that all of `p`'s messages have in fact been delivered back to `p`.

In order for a GCS to be live and satisfy within-view delivery, self-delivery, and virtually synchronous delivery, the GCS must *block* its application from sending new messages during view formation periods; this is proved in [23]. Therefore, we introduce a `block`/`block_ok` synchronization when we extend our algorithm to support the self-delivery property in section 6.3.

Our formulation of self-delivery as a safety property, when combined with the liveness property of section 5.2, implies the formulations in [16] and [38] of self-delivery as a liveness property. These formulations require a GCS to *eventually* deliver to each process its own messages.

5.2. Liveness property. In a fault-prone asynchronous model, it is not feasible to require that a GCS 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, for example, in [22, 17, 16]). Since our GCS uses an external membership service, we condition the GCS liveness on the behavior of the membership service.

We define the liveness property for a restricted set of executions in which a component stabilizes from some point on forever thereafter.

PROPERTY 5.1 (view stability). *Let* GCS *be a group communication service whose interface with its clients consists of* `send`*,* `deliver`*, and* `view` *events as defined in the automaton signature in Figure* 5.1*. Furthermore, assume that the* GCS *uses a membership service* MEMB *described in section* 4*.*

A view v *eventually becomes* stable *in a given timed execution* $\alpha = \mathtt{s}_0, \pi_1, \mathtt{s}_1, \pi_2, \ldots$ *of the* GCS *service, in the sense that a* MEMB.$\mathtt{view}_\mathtt{p}(\mathtt{v})$ *event occurs in* α *for every* p $\in$ v.set *and is followed by neither* MEMB.$\mathtt{view}_\mathtt{p}$ *nor* MEMB.$\mathtt{start}_\mathtt{p}$ *events.*

Given an execution that satisfies Property 5.1, the liveness property requires each end-point in the stable view to eventually deliver this last view and all the messages sent in this view to its client. Formally, we have the following property.

PROPERTY 5.2 (liveness). *Let* GCS *be a group communication service whose interface with its clients consists of* `send`*,* `deliver`*, and* `view` *events as defined in the automaton signature in Figure* 5.1*. Furthermore, assume that the* GCS *uses a membership service* MEMB *described in section* 4*.*

Let α *be a fair execution of* GCS *in which view* v *eventually becomes stable (Property* 5.1*). Then, at each* p $\in$ v.set*,* GCS.$\mathtt{view}_\mathtt{p}(\mathtt{v})$ *eventually occurs. Moreover, for every* GCS.$\mathtt{send}_\mathtt{p}(\mathtt{m})$ *that occurs after* GCS.$\mathtt{view}_\mathtt{p}(\mathtt{v})$*, and for every* q $\in$ v.set*,* GCS.$\mathtt{deliver}_\mathtt{q}(\mathtt{p},\mathtt{m})$ *also occurs.*

It is important to note that although our liveness property requires the GCS to be live only in *certain* executions, any implementation that satisfies this property has to attempt to be live in *every* execution because it cannot test the external condition of the membership becoming stable. Also note that, even though membership stability is formally required to last forever, in practice it only has to hold "long enough" for the GCS to reconfigure, as explained in [21, 25]. However, we cannot explicitly introduce the bound on this time period in a fully asynchronous model, since it depends on external conditions such as message latency, process scheduling, and processing time.

6. The virtually synchronous group multicast algorithm. In this section we present an algorithm for a group communication service, GCS, that satisfies the specifications in section 5. The GCS is implemented by a collection of GCS end-points, each running the same algorithm. Figure 6.1(a) shows the interaction of a GCS end-point with its environment: a membership service MEMB and a reliable FIFO multicast service CO_RFIFO; these services are assumed to satisfy the specifications of section 4. The end-point interacts with its application client by accepting the client's send requests and by delivering application messages and views to the client. The end-point uses the CO_RFIFO service to send messages to other GCS end-points and to receive messages sent by other GCS end-points. When necessary, the end-point uses

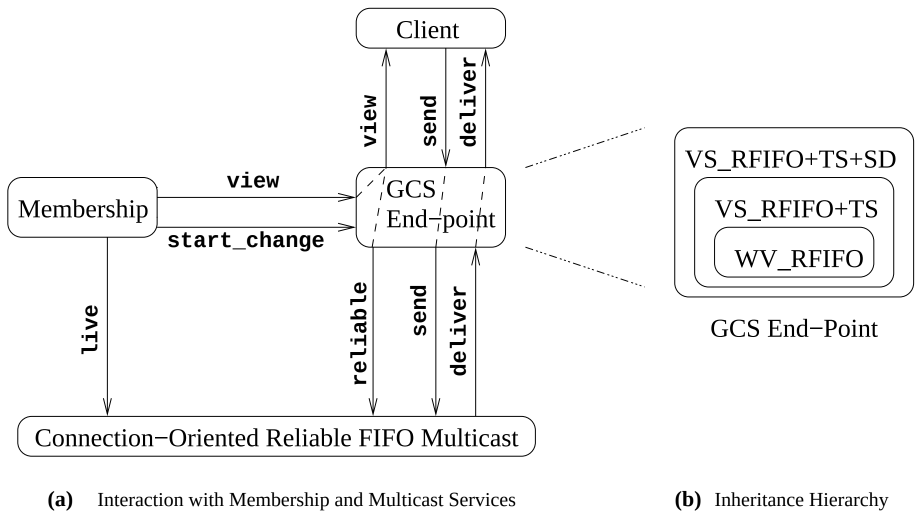

FIG. 6.1. *A GCS end-point and its environment.*

the `reliable` action to inform CO_RFIFO of the set of end-points to which CO_RFIFO must maintain reliable (gap-free) FIFO connections. The GCS end-point also receives `start` and `view` notifications from the membership service.

The algorithm running at each GCS end-point is constructed incrementally using the inheritance-based methodology of [30]. We proceed in three steps, at each step adding support for a new property (see Figure 6.1(b)):

1. In section 6.1, we present an algorithm WV_RFIFO$_p$ for an end-point of the within-view reliable FIFO multicast service specified in section 5.1.1 and argue that this service satisfies safety specification WV_RFIFO : SPEC and liveness Property 5.2.

2. In section 6.2, we add support for the virtually synchronous delivery and transitional set properties specified in sections 5.1.2 and 5.1.3. We present a child VS_RFIFO+TS$_p$ of WV_RFIFO$_p$ and argue that the service built from VS_RFIFO+TS$_p$ end-points satisfies safety specifications VSRFIFO : SPEC and TS : SPEC and liveness Property 5.2.

3. In section 6.3, we add support for the self-delivery property specified in section 5.1.4. The resulting automaton VS_RFIFO+TS+SD$_p$ models a complete GCS end-point. Due to the use of inheritance, the service built from these end-points automatically satisfies safety specifications WV_RFIFO : SPEC, VSRFIFO : SPEC, and TS : SPEC. We argue that it also satisfies safety specification SELF : SPEC and liveness Property 5.2.

In the presented automata, each locally controlled action is defined to be a task by itself, which means that, if it becomes and stays enabled, it eventually gets executed.

When composing automata into a service, actions of the MEMB.$\mathtt{start}_p(\mathtt{id}, \mathtt{set})$ type are linked with CO_RFIFO.$\mathtt{live}_p(\mathtt{set})$, and actions of the MEMB.$\mathtt{view}_p(\mathtt{v})$ type are linked with CO_RFIFO.$\mathtt{live}_p(\mathtt{v.set})$; the "link" operation can be formally expressed using the signature extension construct. When MEMB and CO_RFIFO actions are linked this way, the `live_set`[`p`] variable of CO_RFIFO matches the MEMB's perception of which end-points are alive and connected to `p`. (We assume that every permanently disconnected end-point is eventually excluded by either a `start` or a `view` notification.) In the composed system, all output actions except the application interface are reclassified as internal.

For simplicity of the code, the presented automata do not include certain practical optimizations such as, for example, garbage collection; we point out some of the important ones in section 6.4.

6.1. Within-view reliable FIFO multicast algorithm. In this section we present the WV_RFIFO$_p$ algorithm running at an end-point p of a basic GCS, WV_RFIFO. The end-point algorithm is quite simple: It relies on the MEMB service to form and deliver views involving end-point p; the end-point forwards these views to its client. The algorithm also relies on the CO_RFIFO service to provide reliable gap-free FIFO multicast communication. When the end-point receives a message-send request from its client, it uses CO_RFIFO to send the message to other end-points in the client's current view. The end-point delivers to its client the messages received from other end-points via CO_RFIFO, provided the client's current view matches the views in which the messages were sent. The algorithm keeps track of the views in which messages are sent using the following technique: each time the end-point delivers a view v to its client, it sends a special view_msg message to the end-points in v.set informing them that the end-point's future messages will be sent in view v. Reliable delivery of messages is ensured by having CO_RFIFO maintain a reliable connection to every member of the end-point's view.

Figure 6.2 models the WV_RFIFO$_p$ algorithm as an automaton. The signature defines the interface through which end-point p interacts with its client and with the MEMB and CO_RFIFO services.

When a view v is received from MEMB via action MEMB.view$_p$(v), end-point p saves it in a variable memb_view and then delivers v to its client by executing action view$_p$(v). Variable current_view contains the last view delivered to the client. The precondition, v = memb_view $\neq$ current_view, on the view$_p$(v) action ensures that v is indeed the last view received from MEMB and that it has not already been delivered to the client. After end-point p delivers view v to its client, it sends a view_msg containing v to the rest of the members of current_view.set by using action CO_RFIFO.send$_p$(set, ⟨'**view_msg**', v⟩) with set = current_view.set − {p} and v = current_view. Variable view_msg[p] contains the last view sent as a view_msg. The first precondition on CO_RFIFO.send$_p$(set, ⟨'**view_msg**', v⟩), view_msg[p] $\neq$ current_view, ensures that each view_msg is sent only once, and the second precondition, current_view.set $\subseteq$ reliable_set, ensures that, prior to sending the view_msg, end-point p has requested CO_RFIFO to maintain reliable connection to every member of the client's view by executing action CO_RFIFO.reliable$_p$(set), which sets variable reliable_set to the value of set. When end-point p receives a view_msg from some end-point q via the CO_RFIFO.deliver$_{q,p}$(⟨'**view_msg**', v⟩) action, it stores v in a variable view_msg[q].

End-point p maintains a queue msgs[q][v] per each end-point q and view v; these queues are used for storing application messages received from other end-points via CO_RFIFO.deliver$_{q,p}$ and from the end-point's own client via send$_p$. When action send$_p$(m) occurs, message m is appended to msgs[p][current_view]. The end-point maintains the following indices that enforce message handling in the order of their appearances in the msgs queues:

- last_sent points to the last application message m on msgs[p][current_view] that was sent using CO_RFIFO.send$_p$(set, ⟨'**app_msg**', m⟩);
- last_rcvd[q], for each end-point q, points to the last message m on queue

```
AUTOMATON WV_RFIFO_p

Type:
 ViewMsg = View
 FwdMsg  = Proc × View × AppMsg × Int

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

State:
 // Variables for handling application messages
 For all Proc q, View v: SequenceOf(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 memb_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  memb.view_p(v)
 eff: memb_view ← v

 OUTPUT  view_p(v)
 pre: v =  memb_view ≠ current_view
 eff: current_view ← v
      last_sent  ← 0
      (∀ q) last_dlvrd[q] ← 0

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

 OUTPUT  co_rfifo.send_p(set, ⟨'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(⟨'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]
 eff: last_dlvrd[q] ← last_dlvrd[q] + 1

 OUTPUT  co_rfifo.send_p(set, ⟨'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(⟨'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,⟨'fwd_msg',r,v,m,i⟩)
 pre: (p ∉ set)   ∧   (m = msgs[r][v][i])

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

FIG. 6.2. *Within-view reliable* FIFO *multicast end-point automaton.*

msgs[q][view_msg[q]] that was delivered to p by CO_RFIFO.$\texttt{deliver}_{q,p}$(⟨'**app_msg**', m⟩);

- last_dlvrd[q], for each end-point q, points to the last message m on queue msgs[q][current_view] that was delivered to p's client using CO_RFIFO.$\texttt{deliver}_p$(q, m).

The first precondition of CO_RFIFO.$\texttt{send}_p$(set, ⟨'**app_msg**', m⟩) ensures that a view_msg containing current_view has been already sent to everybody in set = current_view − {p}. The preconditions on sending view_msgs ensure that CO_RFIFO maintains a reliable connection to everyone in set at the time CO_RFIFO.$\texttt{send}_p$(set, ⟨'**app_msg**', m⟩) occurs.

Automaton WV_RFIFO$_p$ also implements auxiliary functionality that allows end-point p to forward an application message received from some end-point to some other end-points. Specifically, using CO_RFIFO.$\texttt{send}_p$(set, ⟨'**fwd_msg**', r, v, m, i⟩), end-point p can forward to some set of end-points the ith message, m, sent by the client at r in view v. In turn, when end-point p receives CO_RFIFO.$\texttt{deliver}_{q,p}$(⟨'**fwd_msg**', r, v, m, i⟩), it stores the forwarded message m in the ith location of the msgs[r][v] queue. The code of WV_RFIFO$_p$ does not specify a particular strategy for forwarding messages; the strategy can be chosen nondeterministically. Such a strategy can be specified by more refined versions of the algorithm and/or by modifications of WV_RFIFO$_p$, as we do in the VS_RFIFO+TS$_p$ modification of the WV_RFIFO$_p$ automaton in section 6.2 below.

Leaving a certain level of nondeterminism at the parent automaton, with the intention of resolving it later at the child automaton, is a technique similar to the use of *abstract methods* or *pure virtual methods* in object-oriented methodology. We use the same technique in the CO_RFIFO.reliable$_p$(set) action when we require set to be a nondeterministic superset of current_view.set. The VS_RFIFO+TS$_p$ modification of WV_RFIFO$_p$ places additional preconditions on this action, thereby specifying precise values for the set argument.

The WV_RFIFO automaton resulting from the composition of all the end-point automata and the MEMB and CO_RFIFO automata models the WV_RFIFO service. The automaton satisfies the safety properties specified by WV_RFIFO : SPEC: it preserves the local monotonicity and self-inclusion properties of view deliveries guaranteed by the MEMB service; it also extends the gap-free FIFO ordered message delivery of CO_RFIFO with the within-view delivery property. The within-view delivery is achieved by delivering messages to the clients only if the views in which the messages were sent match the clients' current views.

Appendix B.1 contains a simulation from WV_RFIFO to WV_RFIFO : SPEC: Actions of automaton WV_RFIFO : SPEC involving view$_p$(v), send$_p$(m), and deliver$_p$(q, m) are simulated when WV_RFIFO takes the corresponding view$_p$(v), send$_p$(m), and deliver$_p$(q, m) actions. The steps of WV_RFIFO involving other actions correspond to empty steps of WV_RFIFO : SPEC. We define the following function R that maps every reachable state s of WV_RFIFO to a reachable state of WV_RFIFO : SPEC, where s[p].var denotes an instance of a variable var of end-point p in a state s:

```
R(s∈ReachableStates(WV_RFIFO)) = t∈ReachableStates(WV_RFIFO : SPEC), where
  for each Proc p, View v:       t.msgs[p][v]   =   s[p].msgs[p][v],
  for each Proc p, Proc q: t.last_dlvrd[p][q]   =   s[q].last_dlvrd[p],
  for each Proc p:          t.current_view[p]   =   s[p].current_view.
```

Lemma B.1 states that R is a refinement mapping from automaton WV_RFIFO to automaton WV_RFIFO : SPEC; the proof relies on a number of invariant assertions, stated and proved in Appendix B.1 as well.

The WV_RFIFO automaton also satisfies liveness (Property 5.2). Consider a fair execution in which each end-point p in v.set receives the same view v from the membership and no view events afterwards. Starting from the time the MEMB.view$_p$(v) action occurs, the view$_p$(v) action stays enabled; therefore it eventually happens due to the fairness of the execution. After view v is delivered to the clients, all messages sent in view v are also eventually delivered to the clients. This is due to the liveness property of CO_RFIFO, which guarantees that messages sent between live and connected end-points (as perceived by the membership service) are eventually delivered to their destinations. We prove these claims formally for the complete GCS algorithm in Appendix C.

6.2. Adding support for virtually synchronous delivery and transitional sets. The WV_RFIFO service of the previous section guarantees that each member p of a view v receives *some* prefix of the FIFO ordered stream of messages sent by every member q in v. In this section, we modify the WV_RFIFO$_p$ algorithm to yield an end-point VS_RFIFO+TS$_p$ of a service, VS_RFIFO+TS, that, in addition to the semantics provided by WV_RFIFO, guarantees that those members that transition from v into

```
AUTOMATON VS_RFIFO+TS_p  MODIFIES  WV_RFIFO_p

Type:  SyncMsg  = StartId × View × (Proc → Int)

Signature Extension:
 Input:  memb.start_p(id, set), StartId 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

 Internal: set_cut_p()  new
```

FIG. 6.3. *Virtually synchronous reliable* FIFO *multicast: signature extension.*

the same view $\mathtt{v}'$ receive not just *some* but *the same* prefix of the message stream sent by each member $\mathtt{q}$ in $\mathtt{v}$. This is the virtually synchronous delivery property, the key property of VS semantics (see section 5.1.2). Overall, the VS_RFIFO+TS service satisfies the VSRFIFO : SPEC and TS : SPEC safety specifications, as well as liveness Property 5.2; we prove these claims, respectively, in Appendixes B.2, B.3, and C.

In a nutshell, here is how VS_RFIFO+TS$_\mathtt{p}$ computes transitional sets and enforces virtually synchronous delivery: When end-point $\mathtt{p}$ is notified via $\mathtt{start_p(cid, set)}$ of the MEMB's attempt to form a new view, $\mathtt{p}$ sends via CO_RFIFO a synchronization message tagged with $\mathtt{cid}$ to every end-point in $\mathtt{set}$. The synchronization message includes $\mathtt{p}$'s current view $\mathtt{v}$ and a mapping $\mathtt{cut}$, such that $\mathtt{cut(q)}$ is the index of the last message from each $\mathtt{q}$ in $\mathtt{v.set}$ that $\mathtt{p}$ commits to deliver in view $\mathtt{v}$.

End-point $\mathtt{p}$ may receive subsequent $\mathtt{start_p(cid, set)}$ notifications from MEMB. When such a notification includes a new $\mathtt{cid}$, $\mathtt{p}$ sends a new synchronization message, with a freshly made cut, to the proposed $\mathtt{set}$; otherwise, when the $\mathtt{cid}$ is the same as the last one, $\mathtt{p}$ simply forwards the last synchronization message to the joining end-points, that is, to the end-points of the current $\mathtt{set}$ that were not listed in the previously proposed membership.

Once $\mathtt{p}$ receives via $\mathtt{view_p(v')}$ a new view $\mathtt{v}'$ from MEMB and a synchronization message tagged with $\mathtt{v'.startId(q)}$ from each end-point $\mathtt{q}$ in $\mathtt{v.set} \cap \mathtt{v'.set}$, $\mathtt{p}$ computes a transitional set from $\mathtt{v}$ to $\mathtt{v}'$ and decides on which messages it needs to deliver to its client in view $\mathtt{v}$ before delivering view $\mathtt{v}'$. A transitional set $\mathtt{T}$ from $\mathtt{v}$ to $\mathtt{v}'$ is computed to include every client $\mathtt{q}$ in $\mathtt{v.set} \cap \mathtt{v'.set}$ whose synchronization message tagged with $\mathtt{v'.startId(q)}$ contains $\mathtt{p}$'s current view $\mathtt{v}$. For each client $\mathtt{r}$ in $\mathtt{v.set}$, end-point $\mathtt{p}$ decides to deliver all the messages of $\mathtt{r}$ that appear in the cut of the synchronization message of any member $\mathtt{q}$ of $\mathtt{T}$. Section 6.2.1 describes two message-forwarding strategies that ensure $\mathtt{p}$'s ability to actually deliver all the messages it decides to deliver. After $\mathtt{p}$ delivers all these messages to its client, it then delivers to its client the new view $\mathtt{v}'$ along with the transitional set $\mathtt{T}$.

Virtually synchronous delivery follows from the fact that all end-points transitioning from view $\mathtt{v}$ to $\mathtt{v}'$ consider the same synchronization messages, compute the same set $\mathtt{T}$, and hence use the same data to decide which messages to deliver in view $\mathtt{v}$ before delivering view $\mathtt{v}'$. Set $\mathtt{T}$ satisfies Definition 5.1 of a transitional set from $\mathtt{v}$ to $\mathtt{v}'$ because (a) every end-point that computes $\mathtt{T}$ is itself included in $\mathtt{T}$, and (b) no end-point $\mathtt{q}$ in $\mathtt{T}$ is allowed to deliver $\mathtt{v}'$ while in some view other than $\mathtt{v}$ because $\mathtt{v'.startId(q)}$ is linked through $\mathtt{q}$'s synchronization message to $\mathtt{v}$.

```
AUTOMATON VS_RFIFO+TS_p  MODIFIES  WV_RFIFO_p

State Extension:
 (StartId × SetOf(Proc))⊥ start, initially ⊥
 For all Proc q, StartId id:  (View v, (Proc→Int) cut)⊥ sync_msg[q][id], initially ⊥
 SetOf(Proc) sync_set, initially empty
 SetOf((Proc × Proc × View × Int)) forwarded_set, initially empty

Transition Restriction:
 INPUT  memb.start_p(cid, set)
 eff: if start ≠ ⊥ ∧ start.id = cid
         then  sync_set ← sync_set ∩ set
         else  sync_set ← ∅
      start ← ⟨cid, set⟩

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

 INTERNAL  set_cut_p()
 pre: start ≠ ⊥ ∧ sync_msg[p][start.id] = ⊥
 eff: Let cut = {⟨q, LongestPrefixOf(msgs[q][current_view])⟩ | q ∈ current_view.set}
      sync_msg[p][start.id] ← ⟨current_view, cut⟩
      sync_set ← {p}

 OUTPUT  co_rfifo.send_p(set, ⟨'sync_msg', cid, v, cut⟩)
 pre: start ≠ ⊥  ∧  sync_msg[p][start.id] ≠ ⊥
      set = (start.set - sync_set) ≠ ∅
      set ⊆ reliable_set
      cid = start.id ∧ ⟨v, cut⟩ = sync_msg[p][cid]
 eff: sync_set ← start.set

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

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

 OUTPUT  view_p(v, T)
 pre: v.startId(p) = start.id                  // to prevent delivery of obsolete views
      v.set - sync_set = ∅                               // all sync msgs are sent
      last_sent ≥ sync_msg[p][v.startId(p)].cut(p)    // sent out your own msgs
      (∀ 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 ← ⊥
      sync_set ← ∅

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

FIG. 6.4. *Virtually synchronous reliable* FIFO *multicast: state extension and transition restriction.*

Figures 6.2, 6.3, and 6.4, together, contain the code of the VS_RFIFO+TS$_p$ automaton that models end-point p of the VS_RFIFO+TS service. Figures 6.3 and 6.4 specify how the WV_RFIFO$_p$ automaton of Figure 6.2 is modified to support virtually synchronous delivery and transitional sets. Figure 6.3 contains signature extension that defines the signatures of new and modified actions; Figure 6.4 contains the state extension and transition restriction defining, respectively, new state variables and new precondition/effect code. We now describe automaton VS_RFIFO+TS$_p$ in detail.

Upon receiving MEMB.start$_p$(cid, set), VS_RFIFO+TS$_p$ stores the cid and set parameters in the id and set fields of a variable start. When start $\neq \perp$, it indicates that VS_RFIFO+TS$_p$ is engaged in a synchronization protocol, during which it exchanges synchronization messages tagged with start.id with the end-points in start.set; after VS_RFIFO+TS$_p$ delivers a view to its client it resets start to $\perp$.

Variable sync_set indicates the set of end-points to which a synchronization message tagged with the latest start.id has already been sent. When end-point p receives start$_p$(cid, set) with a new cid, sync_set is reset to $\emptyset$ to indicate that a new synchronization message needs to be sent to every end-point in set. However, if the cid is the same as the last one, sync_set is set to sync_set $\cap$ set. This way, the end-point will send its last synchronization message only to the joining end-points (i.e., those in set $-$ sync_set), and not to those to which the message was already sent. Notice that the disconnected end-points (i.e., those that are not in set) are removed from sync_set.

After VS_RFIFO+TS$_p$ receives a start$_p$(cid, set) input from MEMB, it executes an internal action, set_cut$_p$(). This action commits p to deliver to its client all the messages it has so far received from the members of its current view. For each member q of current_view.set, cut(q) is set to the length of the longest continuous prefix of messages in msgs[q][current_view].[4] Action set_cut$_p$() results in p's current view being stored in sync_msg[p][start.id].view, the committed cut being stored in sync_msg[p][start.id].cut, and sync_set being set to {p}.

VS_RFIFO+TS$_p$ specifies precise preconditions on the CO_RFIFO.reliable$_p$(set) actions. When VS_RFIFO+TS$_p$ is not engaged in a synchronization protocol (i.e., when start $= \perp$), CO_RFIFO is asked to maintain reliable connection just to the end-points in p's current view, current_view.set. When VS_RFIFO+TS$_p$ is engaged in a synchronization protocol, it requires CO_RFIFO to maintain reliable connection to the members of a forming view, start.set, as well as to those in current_view.set. Thus, CO_RFIFO avoids loss of messages sent to the disconnected end-points in case these end-points are later added to the forming view.

After setting the cut and telling CO_RFIFO to maintain reliable connection to everyone in current_view.set $\cup$ start.set, VS_RFIFO+TS$_p$ uses CO_RFIFO.send$_p$ to send the synchronization message sync_msg[p][start.id] tagged with start.id to the end-points in start.set $-$ sync_set, that is, to all those end-points in the proposed membership to which this synchronization message has not already been sent. Afterwards, sync_set is adjusted to start.set.

When end-point p receives synchronization messages from other end-points, via CO_RFIFO.deliver$_{q,p}$($\langle$'**sync_msg**', cid, v, cut$\rangle$), p saves $\langle$v, cut$\rangle$ in sync_msg[q][cid].

VS_RFIFO+TS$_p$ restricts delivery of application messages while it is engaged in a synchronization protocol (i.e., when start $\neq \perp$ and sync_msg[p][start.id] $\neq \perp$): Prior to receiving a new view from MEMB, only the messages identified in the cut of its own latest synchronization message, sync_msg[p][start.id].cut, can be delivered to the client. After MEMB.view$_p$(v) occurs, VS_RFIFO+TS$_p$ is allowed to deliver messages identified in the cut sync_msg[q][v.startId(q)].cut received from q, provided q is a member of the transitional set from current_view to v. An end-point q $\in$ current_view.set $\cap$ v.set is considered to be in the transitional set from current_view to v if sync_msg[q][v.startId(q)].view is the same as p's current_view.

VS_RFIFO+TS$_p$ delivers a view v received from MEMB and a transitional set T to its client when p has received a synchronization message sync_msg[q][v.startId(q)]

[4] The longest continuous prefix can be different from the length of msgs[q][current_view] because forwarded messages may arrive out of order and introduce gaps in the msgs queues.

from every q in current_view.set $\cap$ v.set, has computed T, and has delivered all the application messages identified in the cuts of the members of T, as specified by the last three preconditions on $\mathtt{view_p(v, T)}$. The first two preconditions ensure, respectively, that no new MEMB.$\mathtt{start_p}$ notification was issued after MEMB.$\mathtt{view_p(v)}$ and that p has sent its synchronization message to everybody in v.set. The third precondition specifies that p has sent to others all of its own messages indicated in its own cut. All these preconditions work in conjunction with those in WV_RFIFO.$\mathtt{view_p(v)}$.

Recall from section 6.1 that WV_RFIFO$_\mathtt{p}$ allows for nondeterministic forwarding of other end-points' application mesages. VS_RFIFO+TS$_\mathtt{p}$ resolves this nondeterminism by placing two additional preconditions on CO_RFIFO.$\mathtt{send_p}$(set, ⟨'**fwd_msg**', r, v, m, i⟩): The first checks a variable forwarded_set to make sure that message m was not previously forwarded to anyone in set. The second tests that a certain predicate, called ForwardingStrategyPredicate(set, r, v, i), holds. This predicate is designed to ensure that all end-points in the transitional set T are able to deliver all the messages that each has committed to deliver in its synchronization message, in particular those sent by disconnected clients. End-points test ForwardingStrategyPredicate to decide whether they need to forward any messages to others.

6.2.1. Forwarding strategy predicate. We now provide two examples of ForwardingStrategyPredicates. With the first, multiple copies of the same message may be forwarded by different end-points. The second strategy reduces the number of forwarded copies of a message. 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 forwarding of messages that have already been forwarded by others.

A simple strategy. With our first strategy, end-point p forwards message m only if p has committed to deliver m. In addition, if m was originally sent in view v, p forwards m to an end-point q only if p does not know of any view of q later than v and if the latest sync_msg from q sent in view v indicates that q has not received message m.

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

If some end-point q is missing a certain message m, then m will be forwarded to q by some end-point p that has committed to deliver m, when p learns from q's synchronization message that q misses m.

Reducing the number of forwarded copies of a message. The second strategy relies on the computed transitional set T from view v to v′ to decide which message should be forwarded by which member of the transitional set. Assume that a member u of T misses a message m that was originally sent in v by a nonmember r of T, but that was committed to delivery by some other members of T. Among these members, ForwardingStrategyPredicate selects the one with the minimal process identifier to forward m to u; variations of this predicate may use a different deterministic rule for selecting a member, for example, accounting for network topology or communication costs. The selected end-point, p, forwards the message to u only if view v′ is the latest view known to p, as specified by the first conjunct below. Otherwise, v′ is an obsolete view, so there is no need to help u transition in to v′. The described strategy does not forward to u ∈ T messages from the members of T because u is guaranteed to receive these messages directly from their original senders (unless v′ becomes obsolete because further view changes occur).

```
ForwardingStrategyPredicate(set, r, v, i) ≡
  Let v′ = memb_view ∧                          // latest view known to {p}
  sync_msg[p][v′.startId(p)] ≠ ⊥ ∧                 // already sent own sync_msg
  Let v = sync_msg[p][v′.startId(p)].view ∧
  (for all q ∈  v.set ∩ v′.set) sync_msg[q][v′.startId(q)] ≠ ⊥  ∧  // received right sync_msgs
  Let T = {q ∈ v.set ∩ v′.set | sync_msg[q][v′.startId(q)].view = v} ∧
  r ∉ T   ∧                           // only forward messages from end-point not in T
  set = {u ∈ T  | sync_msg[u][v′.startId(u)].cut(r) < i }  ∧
  p = min{u ∈ T  | sync_msg[u][v′.startId(u)].cut(r) ≥ i }.
```

If all end-points receive the same view from MEMB, only one copy of m will be forwarded to each u. In rare cases, however, when MEMB delivers different views to different end-points, more than one end-point may forward the same message m to the same end-point u.

Each end-point waits to receive a new view from MEMB and all the right synchronization messages before it forwards messages to others. Thus, compared to the first strategy, this strategy reduces the communication traffic at the cost of slower recovery of lost messages.

6.2.2. Correctness argument. The VS_RFIFO+TS automaton, resulting from the composition of all end-point automata and the MEMB and CO_RFIFO automata, satisfies the VSRFIFO : SPEC and TS : SPEC safety specifications, as well as liveness (Property 5.2), as we formally prove in Appendixes B.2, B.3, and C, respectively. Below we give highlights of these proofs.

VSRFIFO : SPEC is a modification of WV_RFIFO : SPEC. The proof that VS_RFIFO+TS satisfies VSRFIFO : SPEC reuses the proof that WV_RFIFO satisfies WV_RFIFO : SPEC and involves reasoning about only how VSRFIFO : SPEC modifies WV_RFIFO : SPEC. The proof extends refinement mapping R between WV_RFIFO and WV_RFIFO : SPEC with a mapping R_n. R_n maps the cut used by the end-points of VS_RFIFO+TS to move from a view v to a view v′ to the cut[v][v′] variable of VSRFIFO : SPEC. The proof depends on Invariant B.9 and Corollary B.1, which state that all end-points that move from a view v to a view v′ use the same synchronization messages, compute the same transitional set T, and therefore use the same cut.

The proof in Appendix B.3 shows that VS_RFIFO+TS satisfies TS : SPEC. The proof augments VS_RFIFO+TS_p with a *prophecy variable* that guesses, at the time end-point p receives a $\text{start}_p(\text{cid}, \text{set})$ notification from MEMB, possible future views that may contain cid in their startId(p) mappings. For each of these views v′, VS_RFIFO+TS simulates a $\text{set\_prev\_view}_p(v')$ action of TS : SPEC, thereby fixing the previous view of v′ to be p's current view v.

In a fair execution of VS_RFIFO+TS in which the same last view v′ is delivered to all its members and no start events subsequently occur, the three preconditions on the $\text{view}_p(v', T_p)$ delivery are eventually satisfied for every $p \in v'.\text{set}$:

1. Condition $v'.\text{startId}(p) = \text{start.id}$ remains true since the execution has no subsequent start events at p.

2. End-point p eventually receives synchronization messages tagged with the "right" cid from every member of $v.\text{set} \cap v'.\text{set}$ because they keep taking steps towards reliably sending these synchronization messages to p (by low-level fairness of the code) and because CO_RFIFO eventually delivers these messages to p (by the liveness assumption on CO_RFIFO).

3. End-point p eventually receives and delivers all the messages committed to

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

Signature Extension:
Input:  block_ok_p() new                          Output: block_p() new
Internal: set_cut_p() modifies set_cut_p()                view_p(v,T) modifies vs_rfifo+ts.view_p(v,T)
State Extension:
block_status ∈ {unblocked, requested, blocked}, initially unblocked

Transition Restriction:
  INTERNAL  set_cut_p()                           OUTPUT  block_p()
  pre: block_status = blocked                     pre: start ≠ ⊥
                                                       block_status = unblocked
                                                  eff: block_status ← requested

  OUTPUT  view_p(v,T)                             INPUT  block_ok_p()
  eff: block_status ← unblocked                   eff: block_status ← blocked
```

FIG. 6.5. $\mathtt{GCS_p}$ *end-point automaton.*

in the cuts of the members of the transitional set $\mathtt{T_p}$ because for each such message there is at least one end-point in $\mathtt{T_p}$ that has the message in its msgs buffer and that will reliably forward it to p (according to the ForwardingStrategyPredicate) if necessary. Also, p never delivers any messages beyond those committed to in the cuts of the members of $\mathtt{T_p}$ because of the precondition on application message delivery.

6.3. Adding support for self-delivery. As a final step in constructing the automaton that models an end-point of our group communication service, $\mathrm{GCS_p}$, we add support for self-delivery to the $\mathrm{VS\_RFIFO{+}TS_p}$ automaton presented above. Self-delivery requires each end-point to deliver to its client all the messages the client sends in a view, before moving on to the next view.

In order to implement self-delivery, virtually synchronous delivery, and within-view delivery together in a live manner, each end-point must *block* its client from sending new messages while a view change is taking place (as proven in [23]). Therefore, we add to $\mathrm{VS\_RFIFO{+}TS_p}$ an output action block and an input action block_ok. We assume that the client 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 B.4, we formalize this requirement as an abstract client automaton.

The $\mathrm{GCS_p}$ automaton appears in Figure 6.5. After receiving the first start notification in a given view, end-point p issues a block request to its client and awaits receiving a block_ok response before executing $\mathtt{set\_cut_p}()$. As a result of $\mathtt{set\_cut_p}()$, p commits to deliver all the messages its client has sent in the current view. Therefore, p has to deliver all these messages before moving on to a new view, and self-delivery is satisfied. Due to the use of inheritance, the GCS automaton preserves all the safety properties satisfied by its parent. Since end-point p has its own messages on the msgs[p][p] queue and can deliver them to its client, liveness is also preserved. Thus, GCS satisfies all the properties we have specified in section 5.

6.4. Optimizations and extensions. Having formally presented the basic algorithm for an end-point of our virtually synchronous GCS, we now discuss several optimizations and extensions that can be added to the algorithm to make its implementation more practical. Specifically, we discuss ways to reduce the size and number of synchronization messages. (In general, the number of synchronization messages and the size of each message sent during a synchronization protocol can be linear in the number of members.) We also discuss garbage collection and ways to avoid the use of nonvolatile storage.

The first optimization that reduces the size of synchronization messages relies on the following observation: An end-point p does not need to send its current view and its cut to end-points that are not in current_view.set because p cannot be included in their transitional sets. However, these end-points still need to hear from p if p is in their current views. Therefore, end-point p could send a smaller synchronization message to the end-points in start.set − current_view.set, containing its start.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 of this message would know not to include p in their transitional sets for views v' with v'.startId(p) = p's start.id. When using this optimization, p also does not need to include its current view in the synchronization messages sent to current_view.set − start.set, since the view information can be deduced from p's view_msg.

An additional optimization can be used if we strengthen the membership specification to require a MEMB.start with a new identifier to be sent every time MEMB changes its mind about the membership of a forming view. In this case, the latest MEMB.start has the same membership as the delivered MEMB.view. Therefore, the synchronization messages can be shortened to not include information about application messages delivered from end-points in start.set ∩ current_view.set: for an end-point p, end-points that have p in their transitional sets will deliver all the application messages that p sent before its synchronization message.

Other optimizations can reduce the total number of messages sent during synchronization protocol by all end-points. A simple way to do this is to transform the algorithm into a leader-based one, as in [44, 40]. A more scalable approach was suggested by Guo, Vogels, and van Renesse [26]. Their algorithm uses a two-level hierarchy for message dissemination in order to implement VS: end-points send synchronization messages to their designated leaders, which in turn exchange only the cumulative information among themselves. Also, the number of messages exchanged to synchronize multiple groups can be reduced, as suggested in [11, 39], by aggregating information pertaining to multiple groups into a single message.

Another optimization addresses the use of stable storage. Recall that in section 4 we assumed that end-points keep their running states on stable storage, and therefore, recover with their state intact. However, our group multicast service does provide meaningful semantics even when GCS end-points maintain their running state on volatile storage. When an end-point p recovers after a crash, it can start executing with its state reset to an initial value with current_view being the singleton view v_p. It needs to contact the MEMB service to be readmitted to its groups. The client would refrain from sending any messages in its recovered view until it receives a new view from its end-point. This view would satisfy local monotonicity and self-inclusion because these are the properties guaranteed by the MEMB service. The specification of virtually synchronous delivery should be changed so that recovery is interpreted as delivering a singleton view. The remaining safety properties are also preserved because they involve message delivery within a single view.

In a practical implementation of our service, some sort of garbage collection mechanism is required in order to keep the buffer sizes finite. The implementation of [43] discards messages from older views when moving to a new view and also when learning that they were already delivered to every client in the view. This implementation also discards older synchronization messages: an end-point holds on to only the latest synchronization message it has received from each end-point. This optimization does not violate liveness since discarded synchronization messages necessarily pertain to obsolete views.

7. Conclusions. We have designed a novel group multicast service targeted for WANs. Our service implements a variant of the VS semantics that includes a collection of properties that have been shown useful for many distributed applications (see [16]). Many GCSs, for example, [44, 40, 9, 5, 19], support these and similar properties. Our design has been implemented [43] (in C++) as part of a novel architecture for scalable group communication in WANs using the datagram service of [7] and the Moshe membership algorithm [31].

The main contribution of this paper is a VS algorithm run by GCS end-points, in particular, its synchronization protocol, which enforces virtually synchronous delivery. This protocol is invoked when the underlying membership service begins to form a new view, and is run while the view is forming. The protocol involves a single message-exchange round during which members of the forming view send synchronization messages to each other. In contrast to previously suggested VS algorithms (e.g., [23, 5, 26, 3, 9]), our algorithm does not require end-points to preagree upon a globally unique identifier before sending synchronization messages and thus involves less communication. Performing less communication is especially important in WANs, where message latency tends to be high.

Furthermore, unlike the algorithms in [5, 26, 9, 40], our algorithm allows the membership service to change the membership of a forming view while the synchronization protocol is running; the protocol responds immediately to such membership changes.

We are not aware of any other algorithm for VS that does not preagree on common identifiers before sending synchronization messages and that always allows new members to join a forming view while the synchronization protocol is running. Our algorithm achieves these two features by virtue of a simple yet powerful idea: End-points tag their synchronization messages with start identifiers that are locally generated by the membership service; when the membership service forms a view and delivers it to the end-points, the view includes information about which start identifiers were given to which member. This information communicates to the end-points which synchronization messages they need to consider from each member. Since no preagreement upon a common identifier takes place, there is nothing that would inhibit the membership service and the VS algorithm from allowing new members to join the forming view; end-points just have to forward their last synchronization messages to the joiners.

As a second contribution of this paper, our design has demonstrated how to effectively decouple the algorithm for achieving VS from the algorithm for maintaining membership. As argued in [6, 31], such decoupling is important for providing efficient and scalable group communication services in WANs. In previous designs that implement VS atop an external membership service [11, 40], the membership service is not allowed to add new members to an already forming view, and the membership service waits to synchronize with all end-points of the formed view before delivering the view to any of the clients.

A distinct and important characteristic of our design is the high level of formality and rigor at which it has been carried out. This paper has provided precise descriptions of the GCS algorithm and the semantics it provides, as well as a formal proof of the algorithm's correctness—both safety and liveness. Previously, formal approaches based on I/O automata were used to specify the semantics of VS GCSs and to model and verify their applications, for example, in [15, 22, 18, 33, 27]. However, due to their size and complexity, VS algorithms were not previously modeled using formal methods, nor were they assertionally verified. Our experience has taught us the importance of careful modeling and verification: in the process of proving our algorithm's correctness we have often uncovered subtleties and ambiguities that had to be resolved.

In order to manage the complexity of our design, we developed a new formal inheritance-based methodology [30]. The incremental way in which we constructed our algorithms and specifications also allowed us to construct the simulation proof incrementally. For example, in order to prove that VS_RFIFO+TS simulates VS_RFIFO+TS : SPEC, we extended the simulation relation from WV_RFIFO to WV_RFIFO : SPEC and reasoned solely about the extension, without repeating the reasoning about the parent components (see Appendix B.2). This reuse was justified by the proof extension theorem of [30] (see Appendix A.3). The use of incremental construction was the key to our success in formally modeling and reasoning about such a complex and sophisticated service. We hope that the methodology employed in this paper shall also be helpful to other researchers working on formal modeling of complex distributed systems.

Appendix 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 [36, pp. 216–228] and [35, pp. 3, 4, and 13]. In section A.3 we present a proof extension theorem of [30] 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* `traces`*(A)* $\subseteq$ `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:

1. The states of S may contain more information than the states of A.
2. 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 nondeterministic 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 the prophecy variable.

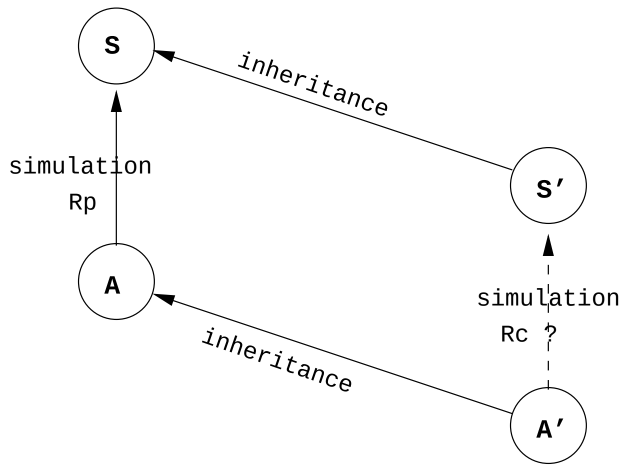

FIG. A.1. *Algorithm* A *simulates specification* S *with* R. *Can* R *be reused for building a refinement* R′ *from a child* A′ *of* A *to a child* S′ *of* S*?*

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 [30] which lays the foundation for incremental proof construction. Consider the example illustrated in Figure A.1, where a refinement mapping R from an algorithm A to a specification S is given, and we want to construct a refinement mapping R′ from a child A′ of an automaton A to a child S′ of a specification automaton S.

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

THEOREM A.2. *Let automaton* A′ *be a child of automaton* A*. Let automaton* S′ *be a child of automaton* S*. Let mapping* R *be a refinement from* A *to* S*.*

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

A mapping R′ *from the states of* A′ *to the states of* S′*, defined in terms of* 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* A′ *to* S′ *if* R′ *satisfies the following two conditions:*

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

1. α *projected onto states of* S *is an execution sequence of* S*.*

2. *Every step* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}) \in \alpha$ *is consistent with the transition restriction placed on* $\mathtt{S}$ *by* $\mathtt{S}'$.
3. *The parent component of the final state* $\mathtt{s}'$ *is* $\mathtt{R}(\mathtt{t}')$.
4. *The child component of the final state* $\mathtt{s}'$ *is* $\mathtt{R_n}(\langle \mathtt{t}', \mathtt{t}'_\mathtt{n} \rangle)$.
5. 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 1, 3, and 5 of Theorem A.2 hold for this sequence. The only conditions that have to be checked are 2, and 4, that is, 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 poststate 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 about 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.

Appendix B. Correctness proof: Safety properties. We now formally prove, using invariant assertions and simulations, that our algorithms satisfy the safety properties of section 5.1. Proofs done with invariant assertions and simulations are verifiable (even by a computer) because they involve reasoning only about single steps of the algorithm. A review of the proof techniques used in this section appears in Appendix A.

The safety proof is *modular*: we exploit the inheritance-based structure of our specifications and algorithms to reuse proofs. In section B.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 B.2 we extend this refinement mapping to map the new state added in VS_RFIFO+TS to that in VSRFIFO : SPEC. In section B.3 we prove that VS_RFIFO+TS also simulates TS : SPEC. Finally, in section B.4 we extend the refinement above to map the new state of GCS to that of SELF : SPEC. The proof extension theorem of [30] (also reviewed in Appendix A) implies that the GCS automaton satisfies WV_RFIFO : SPEC, VSRFIFO : SPEC, TS : SPEC, and SELF : SPEC.

B.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 for delivered views, and we need to show that the ith message delivered by $\mathtt{q}$ from $\mathtt{p}$ in view $\mathtt{v}$ is the ith message sent in view $\mathtt{v}$ by the client at $\mathtt{p}$. In order to prove this, we need to show that the algorithm correctly associates messages with the views in which they were sent and with their indices in the sequences of messages sent in these views. We split the proof into three parts: section B.1.1 states key invariants but defers the proof of one of them to section B.1.3; section B.1.2 contains the simulation proof.

B.1.1. Key invariants. The following invariant captures the self-inclusion property.

INVARIANT B.1 (self-inclusion). *In every reachable state* $\mathtt{s}$ *of* WV_RFIFO*, for all* $\mathtt{Proc\ p}$, $\mathtt{p} \in \mathtt{s[p].memb\_view.set}$ *and* $\mathtt{p} \in \mathtt{s[p].current\_view.set}$.

Proof of Invariant B.1. The proof immediately follows from the MEMB specification. □

The local monotonicity property follows directly from the precondition, $\mathtt{v.id} > \mathtt{memb\_view}$, of the MEMB.$\mathtt{view_p(v)}$ actions.

The following invariant relates application messages at different end-points' queues to the corresponding messages on the original senders' queues.

INVARIANT B.2 (message consistency). *In every reachable state* $\mathtt{s}$ *of* WV_RFIFO*, for all* $\mathtt{Proc\ p}$ *and* $\mathtt{Proc\ q}$*, if* $\mathtt{s[q].msgs[p][v][i]} = \mathtt{m}$*, then* $\mathtt{s[p].msgs[p][v][i]} = \mathtt{m}$.

This proposition is vacuously true in the initial state because all message queues are empty. For the inductive step, we have to consider the CO_RFIFO.$\mathtt{deliver_{q,p}}$ $(\langle\text{‘}\mathbf{app\_msg}\text{’}, \mathtt{m}\rangle)$ and CO_RFIFO.$\mathtt{deliver_{q,p}}(\langle\text{‘}\mathbf{fwd\_msg}\text{’}, \mathtt{r}, \mathtt{v}, \mathtt{m}, \mathtt{i}\rangle)$ actions and have to argue that the message $\mathtt{m}$ that these actions deliver is placed in the right place in $\mathtt{q}$'s $\mathtt{msgs}$ buffer. The proof of this invariant appears in section B.1.3 after the simulation proof.

B.1.2. Simulation.

LEMMA B.1. *The following function* $\mathtt{R}$ *is a* refinement mapping *from automaton* WV_RFIFO *to automaton* WV_RFIFO : SPEC *with respect to their reachable states.*

```
R(s∈ReachableStates(WV_RFIFO)) = t∈ReachableStates(WV_RFIFO : SPEC), where

  For each Proc p, View v:       t.msgs[p][v]   =   s[p].msgs[p][v]

  For each Proc p, Proc q: t.last_dlvrd[p][q]   =   s[q].last_dlvrd[p]

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

Proof of Lemma B.1.

Action correspondence. Automaton WV_RFIFO : SPEC has three types of actions. Actions $\mathtt{view_p(v)}$, $\mathtt{send_p(m)}$, and $\mathtt{deliver_p(q, m)}$ are simulated when WV_RFIFO takes the corresponding $\mathtt{view_p(v)}$, $\mathtt{send_p(m)}$, and $\mathtt{deliver_p(q, m)}$ actions. Steps of WV_RFIFO involving other actions correspond to empty steps of WV_RFIFO : SPEC.

Simulation proof. For 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 $\mathtt{view_p(v)}$ relies on $\mathtt{p} \in \mathtt{memb\_view.set}$ (Invariant B.1).

For the steps involving the $\mathtt{deliver_p(q, m)}$ action, in order to deduce that the corresponding step of WV_RFIFO : SPEC is enabled, we need to know that the message located at index $\mathtt{s[p].last\_dlvrd[q]} + 1$ on the $\mathtt{s[p].msgs[q][s[p].current\_view]}$ queue is the same message that end-point $\mathtt{q}$ has on its corresponding queue at the same index. This property is implied by Invariant B.2.

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 $\mathtt{p}$ to $\mathtt{p}$ because they may affect $\mathtt{s[p].msgs[p][p]}$ queue. Since end-points do not send messages to themselves, such steps may not happen. Indeed, action CO_RFIFO.$\mathtt{send_p(set}, \langle\text{‘}\mathbf{app\_msg}\text{’}, \mathtt{m}\rangle)$ has a precondition $\mathtt{set} = \mathtt{s[p].current\_view.set} - \{\mathtt{p}\}$, and action CO_RFIFO.$\mathtt{send_p(set}, \langle\text{‘}\mathbf{fwd\_msg}\text{’}, \mathtt{r}, \mathtt{v}, \mathtt{m}, \mathtt{i}\rangle)$ has a precondition $\mathtt{p} \notin \mathtt{set}$. □

From Lemma B.1 and Theorem A.1 we conclude the following.

THEOREM B.1. WV_RFIFO *implements* WV_RFIFO : SPEC *in the sense of trace inclusion.*

B.1.3. Auxiliary invariants. We now state and prove a number of auxiliary invariants necessary for the proof of the key message consistency invariant (Invariant B.2).

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 every member of its current view. The following invariant captures this property.

INVARIANT B.3 (connection reliability). *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 of Invariant B.3. By induction on the length of the execution sequence, the proof follows directly from the code. □

After an end-point delivers a new view to its client, it sends a view_msg to other members of the view. The stream of view_msgs that an end-point sends to others is monotonic because the delivered views satisfy local monotonicity. The following invariant captures this property. It states that the subsequence of messages in transit from end-point p to end-point q solely consisting of the view_msgs is monotonically increasing. It also relates the current view of an end-point p to the view contained in the p's latest view_msg to q.

INVARIANT B.4 (monotonicity of view messages). *Let* s *be a reachable state of* WV_RFIFO*. Consider the subsequence of messages in* s.channel[p][q] *of the* ViewMsg *type. Examine the sequence of views included in these view messages, and construct a new sequence* seq *of views by prepending 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 of Invariant B.4. All three propositions are true in the initial state. We now consider steps involving the critical actions.

CO_RFIFO.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 $\notin$ s[p].current_view.set. If it were, the CO_RFIFO.lose(p, q) action would not be enabled because Invariant B.3 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).

$\mathtt{view}_\mathtt{p}$(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] (as can be proved by straightforward induction).

CO_RFIFO.$\mathtt{send}_\mathtt{p}$(set, ⟨'**view_msg**', v⟩). The first proposition is true in the poststate because of the inductive hypothesis of the second proposition. The second proposition is vacuously true in the poststate. The third proposition is true in the poststate because of the effect of this action.

CO_RFIFO.$\mathtt{deliver}_{p,q}$(⟨‘**view_msg**’, v⟩). It is straightforward to see that all three propositions remain true in the poststate. □

History tags. In order to reason about original application messages traveling on CO_RFIFO channels, we need a way to reference, for each of these messages, the view in which it was originally sent and its index in the FIFO ordered sequence of messages sent in that view. To this end, we augment each original application message ⟨‘**app_msg**’, m⟩ with two *history tags*, Hv and Hi, that are set to current_view and last_sent + 1, respectively, when CO_RFIFO.$\mathtt{send}_p$(set, ⟨‘**app_msg**’, m⟩) occurs. (See Appendix A for details on history variables.)

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

With the history tags, the interface between WV_RFIFO and CO_RFIFO for handling original application messages becomes CO_RFIFO.$\mathtt{send}_p$(set, ⟨‘**app_msg**’, m, Hv, Hi⟩) and CO_RFIFO.$\mathtt{deliver}_{p,q}$(⟨‘**app_msg**’, m, Hv, Hi⟩).

The goal of the next three invariants is to show that, when end-point 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.

INVARIANT B.5 (history view consistency). *In every reachable state* s *of* WV_RFIFO*, for all* Procp, Procq*, the following holds. For all messages* ⟨‘**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 of Invariant B.5. The proof follows by induction. A step involving CO_RFIFO.$\mathtt{send}_p$(set, ⟨‘**app_msg**’, m, Hv, Hi⟩) directly follows from part 3 of Invariant B.4. 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. □

The following invariant states that the value of s[p].last_sent equals 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 are already received by q.

INVARIANT B.6. *In every reachable state* s *of* WV_RFIFO*, for all* Procp *and for all* Procq ∈ s[p].current_view.set − {p}*, the following is true.*

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

Proof of Invariant B.6. The proof follows by induction. Consider steps involving the following critical actions.

CO_RFIFO.lose(p, q). Assume that the last message on queue 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, q ∈ s.reliable_set[p], so such a step cannot occur.

We can prove by induction that msg ∈ s.channel[p][q] implies s[p].view_msg[p] = s[p].current_view. By Invariant B.3, s[p].current_view.set ⊆ s[p].reliable_set.

Since $\mathtt{q} \in \mathtt{s}[\mathtt{p}].\mathtt{current\_view.set}$ and $\mathtt{s}[\mathtt{p}].\mathtt{reliable\_set} = \mathtt{s.reliable\_set}[\mathtt{p}]$, it follows that $\mathtt{q} \in \mathtt{s.reliable\_set}[\mathtt{p}]$.

$\mathtt{view_p(v)}$. The proposition remains true for steps involving $\mathtt{view_p(v)}$ action because its effect sets $\mathtt{s'}[\mathtt{p}].\mathtt{last\_sent}$ to $\mathtt{0}$ and because both summands of the right-hand side of the equation also become $\mathtt{0}$. Indeed, the first summand becomes $\mathtt{0}$ because CO_RFIFO channels never have messages tagged with views that are larger than the current views of the messages' senders (as can be shown by a simple inductive proof); the second summand becomes $\mathtt{0}$ because part 2 of Invariant B.4 implies that $\mathtt{s'}[\mathtt{q}].\mathtt{view\_msg}[\mathtt{p}] \neq \mathtt{s'}[\mathtt{p}].\mathtt{current\_view}$.

CO_RFIFO.$\mathtt{deliver_{p,q}}(\langle\text{'}\textbf{view\_msg}\text{'}, \mathtt{v}\rangle)$. The proposition remains true for steps involving this action because $\mathtt{s}[\mathtt{q}].\mathtt{view\_msg}[\mathtt{p}] \neq \mathtt{s}[\mathtt{p}].\mathtt{current\_view}$, as follows immediately from Invariant B.4.

CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle)$ and CO_RFIFO.$\mathtt{deliver_{p,q}}$ $(\langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle)$. For steps involving these actions the truth of the proposition immediately follows from the effects of these actions, the inductive hypotheses, and Invariant B.5. □

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

INVARIANT B.7 (history indices consistency). *In every reachable state* $\mathtt{s}$ *of* WV_RFIFO*, for all* $\mathtt{Proc\ p}$ *and* $\mathtt{Proc\ q}$*, if* $\langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle = \mathtt{s.channel}[\mathtt{p}][\mathtt{q}][\mathtt{j}]$ *for some index* $\mathtt{j}$*, then*

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

Proof of Invariant B.7. In the initial state $\mathtt{s.channel}[\mathtt{p}][\mathtt{q}]$ is empty. For the inductive step, we consider steps involving the following critical actions.

CO_RFIFO.$\mathtt{lose(p,q)}$. The proposition remains true since CO_RFIFO.$\mathtt{lose(p,q)}$ discards only the last messages from the CO_RFIFO $\mathtt{s.channel}[\mathtt{p}][\mathtt{q}]$.

CO_RFIFO.$\mathtt{deliver_{p,q}}(\langle\text{'}\textbf{view\_msg}\text{'}, \mathtt{v}\rangle)$. We have to consider the effects on two types of application messages: those associated with view $\mathtt{s}[\mathtt{q}].\mathtt{view\_msg}[\mathtt{p}]$ and those associated with view $\mathtt{Hv}$. Part 1 of Invariant B.4 and Invariant B.5 imply that there are no application messages with $\mathtt{msg.Hv} = \mathtt{s}[\mathtt{q}].\mathtt{view\_msg}[\mathtt{p}]$ on the CO_RFIFO $\mathtt{channel}[\mathtt{p}][\mathtt{q}]$. Thus, the proposition does not apply for such messages. For those messages that have $\mathtt{msg.Hv} = \mathtt{Hv}$, the proposition remains true because $\mathtt{s'}[\mathtt{q}].\mathtt{last\_rcvd}[\mathtt{p}]$ is set to $\mathtt{0}$ as a result of this action.

CO_RFIFO.$\mathtt{deliver_{p,q}}(\langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle)$. This immediately follows from the effect of this action, the inductive hypothesis, and Invariant B.5.

CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle)$. The inductive step immediately follows from the inductive hypothesis and Invariant B.6. □

We now prove a generalization of Invariant B.2 which relates application messages either in transit on the CO_RFIFO channels or at end-points' queues to their corresponding messages on the senders' queues.

INVARIANT B.8 (general message consistency). *In every reachable state* $\mathtt{s}$ *of* WV_RFIFO*, for all* $\mathtt{Procp}$ *and* $\mathtt{Procq}$*, the following are true.*

1. *If* $\langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle \in \mathtt{s.channel}[\mathtt{p}][\mathtt{q}]$*, then* $\mathtt{s}[\mathtt{p}].\mathtt{msgs}[\mathtt{p}][\mathtt{Hv}][\mathtt{Hi}] = \mathtt{m}$.
2. *If* $\langle\text{'}\textbf{fwd\_msg}\text{'}, \mathtt{r}, \mathtt{m}, \mathtt{v}, \mathtt{i}\rangle \in \mathtt{s.channel}[\mathtt{p}][\mathtt{q}]$*, then* $\mathtt{s}[\mathtt{r}].\mathtt{msgs}[\mathtt{r}][\mathtt{v}][\mathtt{i}] = \mathtt{m}$.

3. *If* $\mathtt{s[q].msgs[p][v][i]} = \mathtt{m}$*, then* $\mathtt{s[p].msgs[p][v][i]} = \mathtt{m}$.

Proof of Invariant B.8. *Basis.* In the initial state all message queues are empty.

Inductive step. The following are the critical actions:

```
send_p(m),
co_rfifo.send_p(set, ⟨'app_msg', m, Hv, Hi⟩),
co_rfifo.deliver_q,p(⟨'app_msg', m, Hv, Hi⟩),
co_rfifo.send_p(set, ⟨'fwd_msg', r, v, m, i⟩),
co_rfifo.deliver_q,p(⟨'fwd_msg', r, v, m, i⟩).
```

For steps involving CO_RFIFO.$\mathtt{deliver}_{\mathtt{q,p}}$(⟨'**app_msg**', m, Hv, Hi⟩), we use Invariant B.5 and Invariant B.7, 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. Inductive steps involving each of the other actions are straightforward. □

Invariant B.2 is a private case of this invariant.

B.2. Virtual synchrony. We now show that automaton VS_RFIFO+TS simulates VSRFIFO : SPEC. We prove this by extending the refinement above using the proof extension theorem of [30] (see Appendix A for details).

B.2.1. Invariants. We prove that end-points that move together from one view to the next consider the same synchronization messages and thus compute the same transitional sets and use the same cuts from the members of the transitional set.

INVARIANT B.9. *In every reachable state* s *of* VS_RFIFO+TS*, for all* Procp*,* Procq*, and for every* StartIdcid*, if* s[q].sync_msg[p][cid] $\neq \perp$*, then* s[q].sync_msg[p][cid] = s[p].sync_msg[p][cid].

Proof of Invariant B.9. The proposition is true in the initial state $\mathtt{s_0}$ as all $\mathtt{s_0}$[q].sync_msg[p][cid] = $\perp$. The inductive step involving a $\mathtt{set\_cut_p}$() action is trivial, for it only affects the case q = p. The inductive step involving a CO_RFIFO.$\mathtt{deliver}_{\mathtt{p,q}}$ (⟨'**sync_msg**', cid, v, cut⟩) action follows immediately from the following proposition:

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

which can be proved by straightforward induction. Indeed, there are two critical actions: CO_RFIFO.$\mathtt{send_p}$(set, ⟨'**sync_msg**', cid, v, cut⟩)—immediate from the code—and CO_RFIFO.$\mathtt{deliver}_{\mathtt{p,p}}$(⟨'**sync_msg**', cid, v, cut⟩)—this may not occur because end-points do not send synchronization messages to themselves. □

COROLLARY B.1. *End-points that move together from one view to the next use the same sets of synchronization messages to calculate transitional sets and message cuts.*

Proof. Consider two end-points that deliver view $\mathtt{v}'$ while in view v. At the time of delivering view $\mathtt{v}'$, each of these end-points has synchronization messages from all end-points in the intersection of these views (second precondition), and these synchronization messages are the same as those at their original end-points (Invariant B.9). Thus, the two end-points calculate the same transitional sets and use the same cuts from the members of this transitional set. □

B.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: ...
     (∀ q ∈ Proc)
       H_cut[current_view][v](q) ← max_{r∈T} (sync_msg[r][v.startId(r)].cut(q)).
```

Variable $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$ is updated every time *any* end-point is delivering view $\mathtt{v}'$ while in view $\mathtt{v}$. Corollary B.1 implies that whenever this happens after $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$ is set for the first time the value of $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$ remains unchanged.

We now extend the refinement mapping $\mathtt{R}()$ of Lemma B.1 with the new mapping $\mathtt{R_n}()$:

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

We call the resulting mapping $\mathtt{R}'()$. We exploit the proof extension theorem of [30] (see Appendix A) in order to prove that $\mathtt{R}'()$ is a refinement mapping from VS_RFIFO+TS to VSRFIFO : SPEC.

LEMMA B.2. *Function* $\mathtt{R}'()$ *defined above is a refinement mapping from automaton* VS_RFIFO+TS *to automaton* VSRFIFO : SPEC.

Proof of Lemma B.2. *Action correspondence.* The action correspondence is the same as that of WV_RFIFO, except for the steps of the type $(\mathtt{s}, \mathtt{view_p}(\mathtt{v}', \mathtt{T}), \mathtt{s}')$ which involve VS_RFIFO+TS 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 $\mathtt{s.H\_cut}[\mathtt{v}][\mathtt{v}'] = \bot$) simulate two steps of VSRFIFO : SPEC: $\mathtt{set\_cut}(\mathtt{v}, \mathtt{v}', \mathtt{s}'.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}'])$ followed by $\mathtt{view_p}(\mathtt{v}')$. The rest (when $\mathtt{s.H\_cut}[\mathtt{v}][\mathtt{v}'] \neq \bot$) simulate single steps that involve just $\mathtt{view_p}(\mathtt{v}')$.

Simulation proof. First, we show that the refinement mapping of WV_RFIFO (presented in Lemma B.1) is still preserved after the modifications introduced by VSRFIFO : SPEC to WV_RFIFO : SPEC. Automaton VSRFIFO : SPEC adds the following preconditions to the $\mathtt{view_p}(\mathtt{v})$ actions of WV_RFIFO : SPEC:

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

Since $\mathtt{set\_cut}(\mathtt{current\_view}[\mathtt{p}], \mathtt{v}, \mathtt{s}'.\mathtt{H\_cut}[\mathtt{current\_view}[\mathtt{p}]][\mathtt{v}])$ is simulated before action $\mathtt{view_p}(\mathtt{v})$, the first precondition holds. The second one follows immediately from the precondition on VS_RFIFO+TS.$\mathtt{view_p}(\mathtt{v}, \mathtt{T})$ and the extended mapping $\mathtt{R}'()$.

Second, we 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 $\mathtt{s}'.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}'] = \mathtt{s.H\_cut}[\mathtt{v}][\mathtt{v}']$, by Corollary B.1. ∎

From Lemmas B.1 and B.2 and from Theorem A.1, we conclude the following.

THEOREM B.2. VS_RFIFO+TS *implements* VSRFIFO : SPEC *in the sense of trace inclusion.*

B.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, but this is sufficient for proving safety properties (see Appendix A).

B.3.1. Invariants.

INVARIANT B.10. *In every reachable state* $\mathtt{s}$ *of* VS_RFIFO+TS*, for all* $\mathtt{Proc\ p}$ *and for all* $\mathtt{StartId\ id}$*, if* $\mathtt{id} > \mathtt{s}[\text{MEMB}].\mathtt{start}[\mathtt{p}].\mathtt{id}$*, then* $\mathtt{s}[\mathtt{p}].\mathtt{sync\_msg}[\mathtt{p}][\mathtt{id}] = \bot$.

Proof of Invariant B.10. The proposition is true in the initial state. It remains true for the inductive step involving MEMB.$\mathtt{start_p(id, set)}$ because $\mathtt{s[memb].start[p].id}$ is increased as a result of this action. For the step involving $\mathtt{set\_cut_p()}$, the proposition remains true because $\mathtt{s[p].start.id} = \mathtt{s[}$MEMB$\mathtt{].start[p].id}$, as implied by the following invariant, which can be proved by straightforward induction.

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

Finally, a step involving CO_RFIFO.$\mathtt{deliver_{q,p}}(\langle$'**sync_msg**'$, \mathtt{cid, v, cut}\rangle)$ does not affect the proposition because the case $\mathtt{q} = \mathtt{p}$ cannot happen since end-points do not send synchronization messages to themselves. □

LEMMA B.3. *For any step* $(\mathtt{s},$ MEMB.$\mathtt{start_p(id, set)}, \mathtt{s'})$ *of* VS_RFIFO+TS,

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

Proof of Lemma B.3. The proof follows from the precondition $\mathtt{id} > \mathtt{s[}$MEMB$\mathtt{].start[p].id}$ and Invariant B.10. □

INVARIANT B.11. *In every reachable state* $\mathtt{s}$ *of* VS_RFIFO+TS, *for all* $\mathtt{Proc p}$, *if* $\mathtt{s[p].start} \neq \perp$ *and* $\mathtt{s[p].sync\_msg[p][s[p].start.id]} \neq \perp$, *then*

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

Proof of Invariant B.11. The proposition is vacuously true in the initial state. For the inductive step, consider the following critical actions:

MEMB.$\mathtt{start_p(id, set)}$. The proposition remains vacuously true because $\mathtt{s'[p].sync\_msg[p][start.id]} = \mathtt{s[p].sync\_msg[p][start.id]} = \perp$ (Lemma B.3).

$\mathtt{set\_cut_p()}$. This follows immediately from the code.

CO_RFIFO.$\mathtt{deliver_{q,p}}(\langle$'**sync_msg**'$, \mathtt{cid, v, cut}\rangle)$. The proposition is unaffected because the case $\mathtt{q} = \mathtt{p}$ cannot happen since end-points do not send synchronization messages to themselves.

$\mathtt{view_p(v)}$. The proposition becomes vacuously true because $\mathtt{s'[p].start} = \perp$. □

B.3.2. Simulation. We augment automaton VS_RFIFO+TS with a prophecy variable $\mathtt{P\_legal\_views(p)(id)}$ for each $\mathtt{Proc p}$ and each $\mathtt{StartId\ id}$. At the time a start $\mathtt{id}$ is delivered to an end-point $\mathtt{p}$, this variable is set to a *predicted* finite set of future views that are allowed to contain $\mathtt{id}$ as $\mathtt{p}$'s start id.

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

INTERNAL  MEMB.start_p(id, set)  hidden parameter V, a finite set of views
pre: ...
     choose V such that for all v ∈ V: (p ∈ v.set) ∧ (v.startId(p) = id)
eff: ...
     P_legal_views(p)(id) ← V

OUTPUT  GCS.view_p(v, T)
pre: ...
     (for all 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 show that the following conditions required for adding a prophecy variable hold:

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

INVARIANT B.12. *In every reachable state* $\mathtt{s}$ *of* VS_RFIFO+TS*, for all* $\mathtt{Proc\ p}$*, if* $\mathtt{s[p].start} \neq \perp$*, then, for all* $\mathtt{View\ v} \in \mathtt{P\_legal\_views(p)(s[p].start.id)}$*, it follows that* $\mathtt{p} \in \mathtt{v.set}$ *and* $\mathtt{v.startId(p)} = \mathtt{s[p].start.id}$*.*

Proof of Invariant B.12. The proof follows by induction. The only critical actions are MEMB.$\mathtt{start_p(id, set)}$ and $\mathtt{view_p(v, T)}$. The proposition is true after the former and is vacuously true after the latter. □

LEMMA B.4. *The following function* $\mathtt{TS()}$ *is a* refinement mapping *from automaton* VS_RFIFO+TS *to automaton* TS : SPEC *with respect to their reachable states.*

$$\mathtt{TS}(\mathtt{s} \in \mathtt{ReachableStates}(\text{VS\_RFIFO+TS})) = \mathtt{t} \in \mathtt{ReachableStates}(\text{TS : SPEC}), \mathtt{where}$$

$$\mathtt{For\ eachProc p:\ t.current\_view[p] = s[p].current\_view}$$

$$\mathtt{For\ eachProc p, View v:\ t.prev\_view[p][v]}$$

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

Proof of Lemma B.4. *Action correspondence*. A step $(\mathtt{s}, \mathtt{set\_cut_p()}, \mathtt{s'})$ of VS_RFIFO+TS simulates a sequence of steps of TS : SPEC. The sequence consists of steps that involve one $\mathtt{set\_prev\_view_p(v')}$ action for each $\mathtt{v'} \in \mathtt{s.P\_legal\_views(p)}$ $\mathtt{(s[p].start.id)}$. A step $(\mathtt{s}, \mathtt{view_p(v, T)}, \mathtt{s'})$ of VS_RFIFO+TS simulates $(\mathtt{TS(s)}, \mathtt{view_p(v, T)}, \mathtt{TS(s')})$ of TS : SPEC.

Simulation proof. Consider the following critical actions:

MEMB.$\mathtt{start_p(id, set)}$. A step involving this action simulates an empty step of TS : SPEC. The simulation holds because $\mathtt{s'[p].sync\_msg[p][id]} = \mathtt{s[p].sync\_msg[p][id]} = \perp$ (Lemma B.3).

$\mathtt{set\_cut_p()}$. This simulates a sequence of steps of TS : SPEC involving one $\mathtt{set\_prev\_view_p(v')}$ for each $\mathtt{v'} \in \mathtt{s.P\_legal\_views(p)(cid)}$, where $\mathtt{cid} = \mathtt{s[p].start.id}$. Each such step is enabled, as can be seen from the following derivation:

$$\begin{aligned} &\mathtt{TS(s).prev\_view[p][v']} \\ &= \mathtt{s[p].sync\_msg[p][v'.startId(p)].view} \text{ (Refinement mapping)} \\ &= \mathtt{s[p].sync\_msg[p][cid].view} \text{ (Invariant B.12)} \\ &= \perp \text{ (precondition of } \mathtt{set\_cut_p()}\text{)}. \end{aligned}$$

In the poststate, $\mathtt{s'[p].sync\_msg[p][cid].view}$ and all $\mathtt{TS(s').prev\_view[p][v']}$ are equal to $\mathtt{s[p].current\_view}$; thus the simulation step holds.

CO_RFIFO.$\mathtt{deliver_{q,p}}(\langle$'**sync_msg**'$, \mathtt{cid}, \mathtt{v}, \mathtt{cut}\rangle)$. 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 $\mathtt{q} = \mathtt{p}$ may not happen because end-points do not send synchronization messages to themselves.

```
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
```

FIG. B.1. *Abstract specification of a blocking client at end-point* p.

$\mathtt{view_p(v,T)}$. A step involving this action simulates a step of TS : SPEC that involves $\mathtt{view_p(v,T)}$. 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 $\mathtt{view_p(v,T)}$ of TS : SPEC has three preconditions. The fact that they are enabled directly follows from the inductive hypothesis, the code, the refinement mapping, and Invariants B.11 and B.12. $\square$

From Lemma B.4 and Theorem A.1 we conclude the following.

THEOREM B.3. VS_RFIFO+TS *implements* TS : SPEC *in the sense of* finite *trace inclusion.*

B.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 B.1. 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.

B.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 B.13. *In every reachable state* s *of* GCS*, for all* Proc p, $\mathtt{s[GCS_p].block\_status = s[client_p].block\_status}$.

Proof of Invariant B.13. The proof follows by trivial induction. $\square$

INVARIANT B.14. *In every reachable state* s *of* GCS*, for all* Procp*, if* $\mathtt{s[p].start \neq \bot}$ *and* $\mathtt{s[p].block\_status \neq blocked}$*, then* $\mathtt{s[p].sync\_msg[p][s[p].start.id] = \bot}$.

Proof of Invariant B.14. In the initial state $\mathtt{s_0}$, $\mathtt{s_0[p].start = \bot}$; so the proposition is vacuously true. For the inductive step, consider the following critical actions:

MEMB.$\mathtt{start_p(id,set)}$. The proposition remains true because of Lemma B.3.

$\mathtt{block_p()}$. The proposition is true in the poststate if it is true in the prestate.

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

$\mathtt{set\_cut_p()}$. The proposition remains vacuously true because $\mathtt{s[p].block\_status = s'[p].block\_status = blocked}$.

CO_RFIFO.$\mathtt{deliver_{q,p}}$(⟨'**sync_msg**', cid, v, cut⟩). The proposition is unaffected because the case q = p cannot happen since end-points do not send synchronization messages to themselves.

$\mathtt{view_p(v, T)}$. The proposition becomes vacuously true because $\mathtt{s'[p].start} = \perp$. ☐

INVARIANT B.15. *In every reachable state* $\mathtt{s}$ *of* GCS*, for all* $\mathtt{Proc\,p}$*, if* $\mathtt{s[p].start} \neq \perp$ *and* $\mathtt{s[p].sync\_msg[p][s[p].start.id]} \neq \perp$*, then* $\mathtt{s[p].sync\_msg[p][s[p].start.id].cut[p]} = \mathtt{LastIndexOf(s[p].msgs[p][s[p].current\_view])}$.

Proof of Invariant B.15. In the initial state $\mathtt{s_0}$, $\mathtt{s_0[p].start} = \perp$, so the proposition is vacuously true. For the inductive step, consider the following critical actions:

$\mathtt{send_p(m)}$. The proposition is vacuously true because $\mathtt{s'[p].sync\_msg[p][s[p].start.id]} = \perp$, as follows from the precondition $\mathtt{s[client_p].block\_status} \neq \mathtt{blocked}$ on this action at $\mathtt{client_p}$, and from Invariants B.13 and B.14.

MEMB.$\mathtt{start_p(id, set)}$. The proposition is vacuously true because $\mathtt{s'[p].sync\_msg[p][id]} = \mathtt{s[p].sync\_msg[p][id]}$, which by Lemma B.3 is $\perp$.

$\mathtt{set\_cut_p()}$. This follows from $\mathtt{p} \in \mathtt{current\_view.set}$ (Invariant B.1) and the precondition $(\mathtt{forall\,q} \in \mathtt{current\_view.set})\ \mathtt{cut(q)} = \mathtt{LongestPrefixOf(msgs[q][v])}$.

CO_RFIFO.$\mathtt{deliver_{q,p}}(\langle$'**sync_msg**'$, \mathtt{cid, v, cut}\rangle)$. The proposition is unaffected because the case $\mathtt{q} = \mathtt{p}$ cannot happen since, as can be proved by straightforward induction, end-points do not send synchronization messages to themselves.

$\mathtt{view_p(v, T)}$. The proposition becomes vacuously true because $\mathtt{s'[p].start} = \perp$. ☐

B.4.2. Simulation. Lemma B.2 in section B.2 on page 117 establishes function $\mathtt{R'}()$ as a refinement mapping from automaton VS_RFIFO+TS to automaton VSRFIFO : SPEC. We now argue that $\mathtt{R'}()$ is also a refinement mapping from automaton GCS to automaton SELF : SPEC.

LEMMA B.5. *Refinement mapping* $\mathtt{R'}()$ *from automaton* VS_RFIFO+TS *to automaton* VSRFIFO : SPEC *(given in Lemma* B.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* $\text{CLIENT}_\mathtt{p}$: SPEC *specification 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 $\mathtt{view_p(v, T)}$ attempts to simulate a step of SELF : SPEC involving $\mathtt{view_p(v)}$. Indeed,

$$\begin{aligned}
\mathtt{s[p].last_dlvrd[p]} &= \mathtt{max_{r\in T} sync_msg[r][v.startId(r)].cut[p]} \text{ (a precondition)}\\
&= \mathtt{s[p].sync_msg[p][v.startId(p)].cut[p]} \text{ (Invariant B.9.)}\\
&= \mathtt{s[p].sync_msg[p][s[p].start.id].cut[p]} \text{ (a precondition)}\\
&= \mathtt{LastIndexOf(s[p].msgs[p][s[p].current_view])} \text{ (Invariant B.15).}
\end{aligned}$$

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

From Lemmas B.1, B.2, and B.5 and Theorem A.1 we conclude the following.

THEOREM B.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* $\text{CLIENT}_\mathtt{p}$: SPEC *specification for blocking clients.*

As a child of VS_RFIFO+TS, GCS also satisfies all the safety properties that VS_RFIFO+TS does, in particular TS : SPEC. Thus, from Theorems B.3 and B.4 we conclude the following.

THEOREM B.5. *Automaton* GCS *implements each of the* WV_RFIFO : SPEC*,*

$\mathrm{VSRFIFO:SPEC}$*,* $\mathrm{TS:SPEC}$*, and* $\mathrm{SELF:SPEC}$ *automata in the sense of trace inclusion, under the assumption that clients at each end-point* $\mathtt{p}$ *satisfy the* $\mathrm{CLIENT}_{\mathtt{p}}:\mathrm{SPEC}$ *specification for blocking clients.*

Appendix C. Correctness proof: Liveness property. In this section we prove that fair executions of our group communication service GCS satisfy liveness property 5.2 of section 5.2. In order to show that a certain action eventually happens, we argue that the preconditions on this action eventually become and stay satisfied, and thus the action eventually occurs, by fairness of the execution. Subsection C.1 below presents a number of invariant that are used in the proof of liveness property 5.2 in subsection C.2.

C.1. Invariants. The following invariant captures the fact that, before an end-point computes who the members of its transitional set are, it does not deliver to its client application messages other than those committed by its own synchronization message. Afterwards, the end-point delivers only the messages committed to delivery by the members of the transitional set.

INVARIANT C.1. *In every reachable state* $\mathtt{s}$ *of* GCS*, for all* $\mathtt{Proc\ p}$*, if* $\mathtt{s[p].start} \neq \bot$ *and* $\mathtt{s[p].sync\_msg[p][s[p].start.id]} \neq \bot$*, then for all* $\mathtt{Proc\ q} \in \mathtt{s[p].current\_view.set}$*,*

1. *if* $\mathtt{s[p].start.id} \neq \mathtt{s[p].memb\_view.startId(p)}$*, then* $\mathtt{s[p].last\_dlvrd[q]} \leq \mathtt{s[p].sync\_msg[p][s[p].start.id].cut[q]}$*;*
2. *otherwise, let* $\mathtt{v} = \mathtt{s[p].current\_view}$*,* $\mathtt{v'} = \mathtt{s[p].memb\_view}$*, 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 of Invariant C.1. The proposition is true in the initial state $\mathtt{s_0}$, since $\mathtt{s_0[p].start} = \bot$. For the inductive step, consider the following critical actions:

$\mathtt{deliver_p(q,m)}$. The proposition remains true because the precondition on this action mimics the statement of this proposition.

$\mathrm{MEMB}.\mathtt{start_p(id,set)}$. The proposition is vacuously true because $\mathtt{s'[p].sync\_msg[p][id]} = \mathtt{s[p].sync\_msg[p][id]}$, which by Lemma B.3 is equal to $\bot$.

$\mathrm{MEMB}.\mathtt{view_p(v)}$. In the poststate, $\mathtt{s[p].start.id} = \mathtt{s[p].memb\_view.startId(p)}$, so we must consider the second proposition. Its truth follows from the inductive hypothesis and the fact that $\mathtt{p} \in \mathtt{T}$, as implied by Invariant B.1.

$\mathtt{set\_cut_p()}$. The proposition holds since index $\mathtt{s[p].last\_dlvrd[q]}$ 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 ($\mathtt{for\ all\ q} \in \mathtt{s[p].current\_view.set}$) $\mathtt{cut(q)} = \mathtt{LongestPrefixOf(s[p].msgs[q][s[p].current\_view])}$.

$\mathrm{CO\_RFIFO}.\mathtt{deliver_{q,p}}(\langle\text{‘}\mathbf{sync\_msg}\text{’}, \mathtt{cid,v,cut}\rangle)$. The proposition is unaffected because the case $\mathtt{q} = \mathtt{p}$ is impossible since end-points do not send cuts to themselves.

$\mathtt{view_p(v,T)}$. The proposition becomes vacuously true because $\mathtt{s'[p].start} = \bot$. ∎

The following invariant states that if an end-point $\mathtt{p}$ has end-point $\mathtt{q}$'s cut committing certain messages sent by end-point $\mathtt{r}$ in view $\mathtt{v}$, then end-point $\mathtt{q}$ has those messages buffered.

INVARIANT C.2. *In every reachable state* $\mathtt{s}$ *of* GCS*, for all* $\mathtt{Proc\ p}$*,* $\mathtt{Proc\ q}$*,* $\mathtt{Proc\ r}$*, and* $\mathtt{StartId\ cid}$*, if* $\mathtt{s[p].sync\_msg[q][cid]} \neq \bot$*, then, for every integer* $\mathtt{i}$ *between* $\mathtt{1}$ *and* $\mathtt{s[p].sync\_msg[q][cid].cut[r]}$*,* $\mathtt{s[q].msgs[r][s[p].sync\_msg[q][cid].view][i]} \neq \bot$*.*

Proof of Invariant C.2. The truth of the invariant follows from Invariant B.9 if we can prove that an end-point's cut commits the end-point to deliver only those messages that it already has on its $\mathtt{msgs}$ queue. Formally, this proposition means that, in every reachable state $\mathtt{s}$ of GCS, for all $\mathtt{Proc\ q}$, if $\mathtt{s[q].start} \neq \perp$ and $\mathtt{s[q].sync\_msg[q][s[q].start.id]} \neq \perp$, then, for all $\mathtt{Proc\ r}$ and all $\mathtt{Int\ i}$ such that $1 \leq \mathtt{i} \leq \mathtt{s[q].sync\_msg[q][s[q].start.id].cut[r]}$, $\mathtt{s[q].msgs[r][s[q].current\_view][i]} \neq \perp$. This proposition can be straightforwardly proved by induction: The only interesting action is $\mathtt{set\_cut_q()}$. The truth of the proposition after this action is taken follows immediately from the precondition (for all $\mathtt{r} \in \mathtt{s[q].current\_view.set}$) $\mathtt{cut(r)} = \mathtt{LongestPrefixOf(s[q].msgs[r][s[q].current\_view])}$. □

INVARIANT C.3. *In every reachable state* $\mathtt{s}$ *of* GCS*, for all* $\mathtt{Proc\ p}$ *and* $\mathtt{Proc\ q}$*, if* $\mathtt{q} \in \mathtt{s[p].sync\_set}$*, then* (a) $\mathtt{q} \in \mathtt{s[p].start.set}$ *and* (b) $\mathtt{q} \in \mathtt{s[p].reliable\_set}$.

Proof of Invariant C.3. The proposition is vacuously true in the initial state, where $\mathtt{s[p].sync\_set}$ is empty. The inductive steps for the critical actions MEMB.$\mathtt{start_p(id, set)}$, GCS.$\mathtt{view_p(v, T)}$, and CO_RFIFO.$\mathtt{send_p(set, \langle}$'**sync_msg**'$\mathtt{, cid, v, cut\rangle)}$ follow immediately from their code in Figure 6.4. The inductive step for the action CO_RFIFO.$\mathtt{reliable\_set_p(set)}$ straightforwardly follows from the precondition-effect code in Figures 6.2 and 6.4. The inductive step for the critical action GCS.$\mathtt{set\_cut_p()}$ follows from the code, which sets $\mathtt{sync\_set}$ to $\{\mathtt{p}\}$, and from the fact that $\mathtt{p}$ is always in its own $\mathtt{reliable\_set}$ and $\mathtt{start.set}$ (provided $\mathtt{start} \neq \perp$), which can be straightforwardly proved by induction. □

C.2. Liveness proof. The following lemma states that, in any execution of GCS, every GCS.$\mathtt{view_p}$ event is preceded by the right MEMB.$\mathtt{view_p}$ event, which itself is preceded by the right MEMB.$\mathtt{start_p}$ event.

LEMMA C.1. *In every execution sequence* α *of* GCS*, the following are true:*

1. *For every* GCS.$\mathtt{view_p(v, T)}$ *event, there is a preceding* MEMB.$\mathtt{view_p(v)}$ *event. Moreover, neither a* MEMB.$\mathtt{start_p}$ *nor a* MEMB.$\mathtt{view_p}$ *event occurs between* MEMB.$\mathtt{view_p(v)}$ *and* GCS.$\mathtt{view_p(v, T)}$.
2. *For every* MEMB.$\mathtt{view_p(v)}$ *event, there is a preceding* MEMB.$\mathtt{start_p(id, set)}$ *event with* $\mathtt{id} = \mathtt{v.startId(p)}$ *and* $\mathtt{set} \supseteq \mathtt{v.set}$ *such that neither a* MEMB.$\mathtt{start_p}$*, nor a* MEMB.$\mathtt{view_p}$ *nor a* GCS.$\mathtt{view_p}$ *event occurs in* α *between* MEMB.$\mathtt{start_p(id, set)}$ *and* MEMB.$\mathtt{view_p(v)}$.

Proof of Lemma C.1.

1. Assume that GCS.$\mathtt{view_p(v, T)}$ occurs in α. Two of the preconditions on GCS.$\mathtt{view_p(v, T)}$ are $\mathtt{v} = \mathtt{p.memb\_view}$ and $\mathtt{v.startId(p)} = \mathtt{p.start.id}$, which can only become satisfied as a result of a preceding MEMB.$\mathtt{view_p(v)}$ event, followed by no MEMB.$\mathtt{start_p}$ and MEMB.$\mathtt{view_p}$ events.
2. Assume that MEMB.$\mathtt{view_p(v)}$ occurs in α. Then a MEMB.$\mathtt{start_p(id, set)}$ event with $\mathtt{id} = \mathtt{v.startId(p)}$ and $\mathtt{set} \supseteq \mathtt{v.set}$ must precede MEMB.$\mathtt{view_p(v)}$ because, by the MEMB specification, it is the only possible event that can cause the preconditions for MEMB.$\mathtt{view_p(v)}$ to become true, and because these preconditions do not hold in the initial state of MEMB.
   There may be several MEMB.$\mathtt{start_p(id, set)}$ events with the same $\mathtt{id}$ and different $\mathtt{set}$ arguments. After the last such event, an occurrence of a different MEMB.$\mathtt{start_p}$ event or a MEMB.$\mathtt{view_p}$ event would violate one of the preconditions of MEMB.$\mathtt{view_p(v)}$; thus, such events may not happen. As a corollary from this and part 1 of this lemma, a GCS.$\mathtt{view_p(v', T')}$ event cannot occur between the last MEMB.$\mathtt{start_p(id, set)}$ and MEMB.$\mathtt{view_p(v)}$. □

LEMMA C.2 (liveness). *Let* α *be a fair execution of a group communication service* GCS *in which view* $\mathtt{v}$ *becomes eventually stable as defined by Property* 5.1. *Then at each end-point* $\mathtt{p} \in \mathtt{v.set}$*,* GCS.$\mathtt{view_p(v, T)}$*, with some* $\mathtt{T}$*, eventually occurs. Furthermore, for every* GCS.$\mathtt{send_p(m)}$ *that occurs after* GCS.$\mathtt{view_p(v, T)}$ *and for every*

$\mathtt{q} \in \mathtt{v.set}$, GCS.$\mathtt{deliver_q(p,m)}$ *also occurs.*

Proof of Lemma C.2. Part I. We first prove that GCS.$\mathtt{view_p(v,T)}$ eventually occurs. Our task is to show that, for each $\mathtt{p} \in \mathtt{v.set}$ and some transitional set $\mathtt{T}$, action GCS.$\mathtt{view_p(v,T)}$ becomes enabled at some point after $\mathtt{p}$ receives MEMB.$\mathtt{view_p(v)}$ 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.$\mathtt{view_p(v,T)}$ is in fact executed.

In order for GCS.$\mathtt{view_p(v,T)}$ to become enabled, its preconditions (see Figures 6.2 and 6.4) must eventually become and stay satisfied until GCS.$\mathtt{view_p(v,T)}$ is executed. We now consider each of these preconditions:

$\mathtt{v = p.memb\_view \neq current\_view}$. This precondition ensures that view $\mathtt{v}$ that is attempted to be delivered to the client at $\mathtt{p}$ is the latest view produced by MEMB and has not yet been delivered to the client. The precondition becomes satisfied as a result of MEMB.$\mathtt{view_p(v)}$. Since in any reachable state of the system MEMB.$\mathtt{memb\_view} =$ $\mathtt{p.memb\_view \geq p.current\_view}$ (local monotonicity), this precondition remains satisfied forever, unless GCS.$\mathtt{view_p(v,T)}$ is executed. This is because, by our assumption, α does not contain any subsequent MEMB.$\mathtt{view_p(v')}$, and, hence, by the contrapositive of part 1 of Lemma C.1, it also does not contain any subsequent GCS.$\mathtt{view_p(v',T')}$ with $\mathtt{v' \neq v}$.

$\mathtt{v.startId(p) = p.start.id}$. This precondition prevents delivery of obsolete views: it ensures that the MEMB service has not issued a new $\mathtt{start}$ notification since the time it produced view $\mathtt{v}$. If this condition is not already satisfied before the last MEMB.$\mathtt{start_p(id,set)}$ event with $\mathtt{id = v.startId(p)}$ and $\mathtt{set \supseteq v.set}$, then it becomes satisfied as a result of this event, which, by part 1 of Lemma C.1, must precede MEMB.$\mathtt{view_p(v)}$ in α.

This condition stays satisfied from the time of the last MEMB.$\mathtt{start_p(id,set)}$ at least until GCS.$\mathtt{view_p(v,T)}$ occurs because the only two types of actions, MEMB.$\mathtt{start_p(id',set')}$ and GCS.$\mathtt{view_p(v',T')}$ with $\mathtt{v' \neq v}$, that may affect the value of $\mathtt{p.start}$ cannot occur in α after MEMB.$\mathtt{start_p(id,set)}$, as implied by the assumption on this lemma and Lemma C.1.

$\mathtt{v.set - sync\_set} = \emptyset$. This precondition ensures that prior to delivering view $\mathtt{v}$, end-point $\mathtt{p}$ sends out its synchronization message to every member of $\mathtt{v}$.

Notice that if this precondition becomes satisfied any time after the occurrence of the last MEMB.$\mathtt{start_p(id,set)}$ event with $\mathtt{id = v.startId(p)}$ and $\mathtt{set \supseteq v.set}$, then it stays satisfied from then on until GCS.$\mathtt{view_p(v,T)}$ is executed. If the precondition is not already satisfied right after the MEMB.$\mathtt{start_p}$ action, it becomes satisfied as a result of CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \langle$'**sync_msg**'$, \mathtt{v.startId(p), v, cut}\rangle)$ with $\mathtt{set = p.start.set}$ $-\ \mathtt{p.sync\_set}$. This CO_RFIFO.$\mathtt{send_p}$ action must eventually occur in α because its two preconditions, $(\mathtt{p.sync\_msg[p][id]} \neq \perp)$ and $(\mathtt{set \subseteq reliable\_set})$, eventually become satisfied, for the following reasons.

1. If the first precondition holds any time after the last MEMB.$\mathtt{start_p(id,set)}$ event with $\mathtt{id = v.startId(p)}$ and $\mathtt{set \supseteq v.set}$ occurs, then it stays satisfied from that point on. If it is not already satisfied right after the MEMB.$\mathtt{start_p}$ action, it becomes satisfied as a result of $\mathtt{set\_cut_p()}$. In order for $\mathtt{set\_cut_p()}$ to occur, its precondition, $\mathtt{block\_status = blocked}$, has to becomes satisfied (see Figure 6.5). This occurs as a result of a $\mathtt{block\_ok_q()}$ input from the client at $\mathtt{q}$. If $\mathtt{block\_status}$ equals $\mathtt{blocked}$ at anytime after MEMB.$\mathtt{start_q(v.startId(q),set)}$, then it remains such until GCS.$\mathtt{view_q(v)}$ happens because $\mathtt{block_q()}$ is not enabled after that, and because GCS.$\mathtt{view_q(v)}$ is the only possible GCS view event (by the contrapositive of part 1 of Lemma C.1). To see that $\mathtt{block\_status}$ does in fact become $\mathtt{blocked}$, consider the

three possible values of $\mathtt{block\_status}$ right after MEMB.$\mathtt{start_q}(\mathtt{v.startId(q), set})$ occurs:

1. $\mathtt{block\_status} = \mathtt{blocked}$: We are done.
2. $\mathtt{block\_status} = \mathtt{requested}$: By Invariant B.13, $\mathtt{client.block\_ok_q}()$ 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. $\mathtt{block\_status} = \mathtt{unblocked}$: When MEMB.$\mathtt{start_q}(\mathtt{v.startId(q), set})$ occurs, $\mathtt{block_q}()$ becomes and stays enabled until it is executed. After that, $\mathtt{block\_status}$ becomes $\mathtt{requested}$ and the same reasoning as in the previous case applies.

2. The second precondition, $\mathtt{set} \subseteq \mathtt{reliable\_set}$, becomes satisfied as a result of action CO_RFIFO.$\mathtt{reliable_q}(\mathtt{set})$ with $\mathtt{set} = \mathtt{current\_view.set} \cup \mathtt{start.set}$. This action becomes enabled when $\mathtt{q}$ receives MEMB.$\mathtt{start_q}(\mathtt{v.startId(q), set})$, and therefore it eventually occurs. Afterwards, $\mathtt{reliable\_set}$ remains unchanged because CO_RFIFO.$\mathtt{reliable_q}(\mathtt{set})$ remains disabled; this is because of the precondition $\mathtt{reliable\_set} \neq \mathtt{set}$ and the fact that $\mathtt{q}$'s $\mathtt{current\_view}$ and $\mathtt{start}$ remain unchanged.

When CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \langle\text{'}\mathbf{sync\_msg}\text{'}, \mathtt{v.startId(p), v, cut}\rangle)$ occurs, $\mathtt{p.sync\_set}$ is set to $\mathtt{p.start.set}$. Since $\mathtt{v.set}$ is a subset of $\mathtt{p.start.set}$, this implies that $\mathtt{v.set} - \mathtt{p.sync\_set}$ eventually becomes and stays $\emptyset$.

$(\mathtt{for\ all\ q} \in \mathtt{v.set} \cap \mathtt{p.current\_view.set})\ \mathtt{p.sync\_msg[q][v.startId(q)]} \neq \bot$. This precondition ensures that $\mathtt{p}$ has received the right synchronization message from every $\mathtt{q}$ in $\mathtt{v.set} \cap \mathtt{p.current\_view.set}$. The argument above implies that $\mathtt{q}$ eventually sends to $\mathtt{p}$ a synchronization message tagged with $\mathtt{v.startId(q)}$ and, at the same time, adds $\mathtt{p}$ to $\mathtt{q.sync\_set}$, where $\mathtt{p}$ remains forever, unless GCS.$\mathtt{view_p}(\mathtt{v, T})$ with some $\mathtt{T}$ occurs. In order to conclude that CO_RFIFO eventually delivers this synchronization message to $\mathtt{p}$, we argue that, from the time the last synchronization message from $\mathtt{q}$ to $\mathtt{p}$ is placed on CO_RFIFO.$\mathtt{channel[q][p]}$ and at least until it is delivered to $\mathtt{p}$, end-point $\mathtt{p}$ is in both CO_RFIFO.$\mathtt{reliable\_set[q]}$ and CO_RFIFO.$\mathtt{live\_set[q]}$. The former implies that CO_RFIFO does not lose any messages (in particular, this synchronization message) from $\mathtt{q}$ to $\mathtt{p}$. In conjunction with α being a fair execution, the latter implies that CO_RFIFO eventually delivers every message (in particular, this synchronization message) on the channel from $\mathtt{q}$ to $\mathtt{p}$.

1. From the time $\mathtt{q}$ sends to $\mathtt{p}$ the last synchronization message tagged with $\mathtt{v.startId(q)}$ until GCS.$\mathtt{view_q}(\mathtt{v, T})$ occurs, $\mathtt{p}$ is included in $\mathtt{q.sync\_set}$. Invariant C.3 implies that in that period $\mathtt{p}$ is included in CO_RFIFO.$\mathtt{reliable\_set[q]}$. After GCS.$\mathtt{view_q}(\mathtt{v, T})$ occurs, $\mathtt{p}$ is still included in CO_RFIFO.$\mathtt{reliable\_set[q]}$, since $\mathtt{p} \in \mathtt{v.set}$.

2. End-point $\mathtt{p}$ becomes a member of CO_RFIFO.$\mathtt{live\_set[q]}$ at the time of MEMB.$\mathtt{view_q}(\mathtt{v})$, because MEMB.$\mathtt{view_q}(\mathtt{v})$ is linked to CO_RFIFO.$\mathtt{live\_set_q}(\mathtt{v.set})$ and because $\mathtt{p} \in \mathtt{v.set}$. This property remains true afterward because α does not contain any subsequent MEMB events at end-point $\mathtt{q}$.

Thus, end-point $\mathtt{p}$ eventually receives the right synchronization messages from every $\mathtt{q}$ in $\mathtt{v.set} \cap \mathtt{p.current\_view.set}$.

$\mathtt{last\_sent} \geq \mathtt{sync\_msg[p][v.startId(p)].cut(p)}$. This precondition ensures that before delivering view $\mathtt{v}$, $\mathtt{p}$ sends to others all of its own messages indicated in its own cut. This precondition eventually becomes satisfied because sending of application messages via CO_RFIFO.$\mathtt{send_p}$, which increments $\mathtt{p.last\_sent}$, is enabled at least until $\mathtt{p.last\_sent}$ reaches $\mathtt{sync\_msg[p][v.startId(p)].cut(p)}$, as implied by Invariant C.2.

(for all q$\in$current_view.set) p.last_dlvrd[q]$=$max$_{r\in T}$ p.sync_msg[r][v.startId (r)].cut[q]. This precondition verifies that p has delivered to its client exactly the application messages that it needs to deliver in order for virtually synchronous delivery to be satisfied. By Invariant C.1, the value of p.last_dlvrd[q] never exceeds max$_{r\in T}$ {p.sync_msg[r][v.startId(r)].cut[q]} for any q. It is therefore left to show that p.last_dlvrd[q] does not remain smaller than 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 p.last_dlvrd[q] $<$ max$_{r\in T}$p.sync_msg[r][v.startId(r)].cut[q], and let i be p.last_dlvrd[q] $+$ 1. We now argue that p.last_dlvrd[q] eventually becomes i, that is, that p eventually delivers the next message from q. An inductive application of this argument would imply that p.last_dlvrd[q] eventually reaches max$_{r\in T}$ {p.sync_msg[r][v.startId(r)].cut[q]}.

All the preconditions (except perhaps p.msgs[q][p.current_view][i] $\neq \perp$) for delivering the ith message from q are eventually satisfied because they are the same as the preconditions for p delivering view v, which we have shown to be satisfied. Thus, if the ith message is already on p.msgs[q][p.current_view][i], then delivery of this message eventually occurs by fairness, resulting in p.last_dlvrd[q] being incremented; in this case, we are done.

Therefore, consider the case when p lacks the ith message, m, from q. There are two possibilities:

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

1. q's view prior to installing view v is the same as p's current view (by definition of T and Invariant B.11).
2. q's reliable_set contains p starting before q sent any messages in that view and continuing for the rest of α.
3. Invariant C.2 implies that q has this message and all the messages that precede it in q.msgs[q][p.current_view].
4. End-point q is enabled to send these messages to p in FIFO order. The only event that could prevent q from sending these messages is GCS.view$_q$(v), as it would change the value of q.current_view. However, as we argued above, q must send all of the messages it committed in its cut before delivering view GCS.view$_q$(v). Self-delivery (Invariant B.15) implies that q's cut includes all of the messages q sent while in v. Thus, q would eventually send m to p.
5. The fact that the connection between end-points q and p is live at least after MEMB.view$_q$(v) occurs implies that CO_RFIFO eventually delivers this message to p.

2. 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 to deliver the ith 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:

1. Invariant C.2 implies that r has this message on its r.msgs[r][p.current_view] queue.
2. r's reliable_set contains p starting before r sent any messages in that view and continuing for the rest of α.
3. Upon examination of each of the ForwardingStrategyPredicates in section 6.2.1,

we see that the preconditions for r forwarding the i'th message of q to a set including p eventually become and stay satisfied.

4. Since in both forwarding strategies there is only a finite number of messages from q sent in this view that can be forwarded, fairness implies that the i's message is eventually forwarded to p.
5. The fact that the connection between r and p is live at least after MEMB.$\mathtt{view_q(v)}$ occurs implies that CO_RFIFO eventually delivers this message to p.

Therefore, the ith message from q is eventually delivered to end-point p, and since, as a result of this, the preconditions on delivering this message to the client at p are satisfied, this delivery eventually occurs, and p.last_dlvrd[q] is incremented. By 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 p delivering GCS.$\mathtt{view_p(v, T)}$ eventually becomes and stays satisfied. Fairness implies that GCS.$\mathtt{view_p(v, T)}$ eventually occurs.

Part II. We now consider the second part of the lemma. The following argument proves that, after GCS.$\mathtt{view_p(v, T)}$ occurs at p, for every subsequent GCS.$\mathtt{send_p(m)}$ event at p, there is a corresponding GCS.$\mathtt{deliver_q(p, m)}$ event that occurs at every q $\in$ v.set:

1. For the rest of α, after GCS.$\mathtt{view_p(v, T)}$ occurs, CO_RFIFO.live_set[p] is equal to v.set. This is true because CO_RFIFO.live_set[p] is set to v.set when MEMB.$\mathtt{view_p(v)}$ occurs and remains unchanged thereafter because of the assumption that α does not contain any subsequent MEMB events at end-point p.
2. After GCS.$\mathtt{view_p(v, T)}$ occurs and before any CO_RFIFO.$\mathtt{send_p}$ event involving a ViewMsg or an AppMsg occurs, p eventually executes CO_RFIFO.$\mathtt{reliable_p(v.set)}$. Moreover, after that and forever thereafter, both p.reliable_set and CO_RFIFO.reliable_set[p] equal v.set. This is true because GCS.$\mathtt{view_p(v, T)}$ sets p.start to $\perp$ and p.current_view.set to v.set, thus enabling CO_RFIFO.$\mathtt{reliable_p(v.set)}$. This action eventually happens because α is a fair execution and because for the rest of α there are no subsequent MEMB.$\mathtt{start_p}$ and GCS.$\mathtt{view_p(v', T')}$ events. Because of the latter reason, p.start and p.current_view.set remain unchanged. Therefore, CO_RFIFO.$\mathtt{reliable_p}$ remains disabled and both variables CO_RFIFO.reliable_set[p] and p.reliable_set remain equal to v.set.
   From the above argument and from fairness, it follows that any kind of message that end-point p sends subsequently to q via CO_RFIFO will eventually reach end-point q.
3. Action CO_RFIFO.$\mathtt{send_p}(\mathtt{v.set} - \{p\}, \langle\text{'}\textbf{view\_msg}\text{'}, v\rangle)$ eventually occurs after action CO_RFIFO.$\mathtt{reliable_p(v.set)}$ occurs, as follows from the code in Figure 6.4. By the reasoning above, CO_RFIFO delivers this ViewMsg to every end-point q $\in$ v.set $- \{p\}$, resulting in q.view_msg[p] being set to v for the remainder of α (Invariant B.4).
4. When GCS.$\mathtt{send_p(m)}$ event occurs at p, m is appended to p.msgs[p][v].
5. After sending the ViewMsg, for the rest of α, if p.msgs[p][v][p.last_sent + 1] contains a message (say m′), action CO_RFIFO.$\mathtt{send_p}(\mathtt{v.set} - \{p\}, \langle\text{'}\textbf{app\_msg}\text{'}, m'\rangle)$ is enabled, and hence eventually occurs by fairness. Since p.last_sent is incremented after each application message is sent using CO_RFIFO.$\mathtt{send_p}$, any message on p.msgs[p][v] is eventually sent to v.set $- \{p\}$. As was argued above, these messages are eventually delivered to every end-point q $\in$

$\texttt{v.set} - \{\texttt{p}\}$. Since $\texttt{q.view\_msg}[\texttt{p}] = \texttt{v}$ at the time $\texttt{q}$ receives $\texttt{m}'$, $\texttt{q}$ puts $\texttt{m}'$ in $\texttt{q.msgs}[\texttt{p}][\texttt{v}][\texttt{q.last\_rcvd} + 1]$ (Invariant B.5) and increments $\texttt{q.last\_rcvd}$. Therefore, all messages that end-point $\texttt{p}$ sends in view $\texttt{v}$ are eventually inserted with no gaps in the end-point $\texttt{q}$'s queue, $\texttt{q.msgs}[\texttt{p}][\texttt{v}]$, for every $\texttt{q} \in \texttt{v.set} - \{\texttt{p}\}$.

6. Once $\textsc{gcs}.\texttt{view}_\texttt{q}(\texttt{v}, \texttt{T})$ happens (by part I of the proof of the lemma), end-point $\texttt{q} \in \texttt{v.set}$ is continuously enabled to deliver a message, $\texttt{m}'$, from $\texttt{q.msgs}[\texttt{p}][\texttt{v}][\texttt{q.last\_dlvrd} + 1]$; by fairness, such delivery eventually occurs, resulting in $\texttt{q.last\_dlvrd}[\texttt{p}]$ being incremented. Therefore, every messages on $\texttt{q.msgs}[\texttt{p}][\texttt{v}]$ is eventually delivered to the client at $\texttt{p}$, including the case of $\texttt{q} = \texttt{p}$.

It follows from this argument that every $\textsc{gcs}.\texttt{send}_\texttt{p}(\texttt{m})$ event at end-point $\texttt{p}$ that occurs after $\textsc{gcs}.\texttt{view}_\texttt{p}(\texttt{v}, \texttt{T})$ in α is eventually followed by a $\textsc{gcs}.\texttt{deliver}_\texttt{q}(\texttt{p}, \texttt{m})$ at every $\texttt{q} \in \texttt{v.set}$. $\square$

Acknowledgments. We thank Alan Fekete, Nancy Lynch, Alex Shvartsman, Jeremy Sussman, and the anonymous referee for helpful suggestions.

REFERENCES

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

[2] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev, *Group communication as an infrastructure for distributed system management*, in Proceedings of the 3rd International Workshop on Services in Distributed and Networked Environment (SDNE '96), 1996, IEEE Computer Society, pp. 84–91.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki, *Transis: A communication sub-system for high availability*, in Proceedings of the 22nd Symposium on Fault-Tolerant Computing (FTCS '92), 1992, IEEE Computer Society, pp. 76–89.

[4] 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, Hebrew University, Jerusalem, Israel, 1994.

[5] 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 (1995), pp. 311–342.

[6] T. Anker, G. Chockler, D. Dolev, and I. Keidar, *Scalable group membership services for novel applications*, in Networks in Distributed Computing, M. Mavronicolas, M. Merritt, and N. Shavit, eds., DIMACS Ser. Discrete Math. Theoret. Comput. Sci., AMS, Providence, RI, 1998, pp. 23–42.

[7] T. Anker, G. Chockler, I. Shnaiderman, and D. Dolev, *The Design of Xpand: A Group Communication System for Wide Area Networks*, Technical report 2000-31, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 2000.

[8] T. Anker, D. Dolev, and I. Keidar, *Fault tolerant video-on-demand services*, in Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99), 1999, pp. 244–252.

[9] Ö. Babaoğlu, R. Davoli, and A. Montresor, *Group communication in partitionable systems: Specification and algorithms*, IEEE Trans. Software Engrg., 27 (2001), pp. 308–336.

[10] K. Birman, *Building Secure and Reliable Network Applications*, Manning, Greenwich, CT, 1996.

[11] K. Birman, R. Friedman, M. Hayden, and I. Rhee, *Middleware support for distributed multimedia and collaborative computing*, Software Practice and Experience, 29 (1999), pp. 1285–1312.

[12] K. Birman and T. Joseph, *Exploiting virtual synchrony in distributed systems*, in Proceedings of the 11th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), 1987, pp. 123–138.

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

[14] G. Chockler, N. Huleihel, and D. Dolev, *An adaptive totally ordered multicast protocol that tolerates partitions*, in Proceedings of the 17th ACM Symposium on Principles of

Distributed Computing, 1998, pp. 237–246.

[15] G. V. Chockler, *An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions*, Masters thesis, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1997.

[16] G. V. Chockler, I. Keidar, and R. Vitenberg, *Group communication specifications: A comprehensive study*, ACM Comput. Surveys, 33 (2001), pp. 1–43.

[17] 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, CA, 1995.

[18] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman, *A dynamic view-oriented group communication service*, in Proceedings of the 17th ACM Symposium on Principles of Distributed Computing, 1998, pp. 227–236.

[19] D. Dolev and D. Malkhi, *The Transis approach to high availability cluster communication*, Comm. ACM, 39 (1996), pp. 64–70.

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

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

[22] A. Fekete, N. Lynch, and A. Shvartsman, *Specifying and using a partitionable group communication service*, ACM Trans. Comput. Syst., 19 (2001), pp. 171–216.

[23] R. Friedman and R. van Renesse, *Strong and Weak Virtual Synchrony in Horus*, Technical report TR 95-1537, Department of Computer Science, Cornell University, Ithaca, NY, 1995.

[24] R. Friedman and A. Vaysburd, *Fast replicated state machines over partitionable networks*, in Proceedings of the 16th IEEE International Symposium on Reliable Distributed Systems (SRDS '97), 1997, pp. 130–137.

[25] 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), 1997, IEEE Computer Society, 1997, pp. 183–188.

[26] K. Guo, W. Vogels, and R. van Renesse, *Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication*, in Proceedings of the 7th ACM SIGOPS European Workshop, 1996, pp. 213–217.

[27] J. Hickey, N. Lynch, and R. van Renesse, *Specifications and proofs for ensemble layers*, in Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), Lecture Notes in Comput. Sci. 1579, Springer–Verlag, New York, 1999.

[28] M. Hiltunen and R. Schlichting, *Properties of membership services*, in Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems, 1995, IEEE Computer Society, pp. 200–207.

[29] I. Keidar and D. Dolev, *Efficient message ordering in dynamic networks*, in Proceedings of the 15th ACM Symposium on Principles of Distributed Computing, 1996, pp. 68–76.

[30] I. Keidar, R. Khazan, N. Lynch, and A. Shvartsman, *An inheritance-based technique for building simulation proofs incrementally*, ACM Trans. Software Engrg. and Methodology, 11 (2002), pp. 1–29.

[31] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev, *Moshe: A group membership service for WANs*, ACM Trans. Comput. Syst., 20 (2002), pp. 191–238.

[32] R. Khazan, *A One-Round Algorithm for Virtually Synchronous Group Communication in Wide Area Networks*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 2002.

[33] R. Khazan, A. Fekete, and N. Lynch, *Multicast group communication as a base for a load-balancing replicated data service*, in Proceedings of the 12th International Symposium on Distributed Computing (DISC), Springer-Verlag, New York, 1998, pp. 258–272.

[34] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Comm. ACM, 21 (78), pp. 558–565.

[35] B. Lampson, *Generalizing Abstraction Functions*, Principles of Computer Systems class, Handout 8, Laboratory for Computer Science, MIT, Cambridge, MA, 1997; Available via anonymous ftp from ftp://theory.lcs.mit.edu/pub/classes/6.826/www/6.826-top.html.

[36] N. A. Lynch, *Distributed Algorithms*, Morgan–Kaufmann, San Francisco, 1996.

[37] N. A. Lynch and M. Tuttle, *An introduction to input/output automata*, CWI Quarterly, 2 (1989), pp. 219–246.

[38] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, *Extended Virtual Synchrony*, Technical report ECE93-22, Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA, 1994.

[39] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, and K. Birman, *A dynamic light-weight group service*, in Proceedings of the 15th IEEE International Symposium on Reliable Distributed Systems (SRDS '96), 1996, pp. 23–25; Technical report TR96-1611, Cornell University, Ithaca, New York, 1996.

[40] A. Schiper and A. Ricciardi, *Virtually synchronous communication based on a weak failure suspector*, in Proceedings of the 23rd IEEE Fault-Tolerant Computing Symposium (FTCS '93), 1993, pp. 534–543.

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

[42] J. Sussman and K. Marzullo, *The* bancomat *problem: An example of resource allocation in a partitionable asynchronous system*, Theoret. Comput. Sci., to appear.

[43] I. Tarashchanskiy, *Virtual Synchrony Semantics: Client-Server Implementation*, Master of engineering masters thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 2000.

[44] R. van Renesse, K. P. Birman, and S. Maffeis, *Horus: A flexible group communication system*, Comm. ACM, 39 (1996), pp. 76–83.