20th ICDS, pp. 344–355, April 2000
(Full version: MIT LCS Technical Report MIT-LCS-TR-794).

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

Idit Keidar and Roger Khazan

MIT Laboratory for Computer Science
{idish, roger}@theory.lcs.mit.edu

Abstract. This paper presents a formal design for a novel group multicast service that provides virtually synchronous semantics in asynchronous fault-prone environments. The design employs a client-server architecture in which group membership is maintained not by every process but only by dedicated membership servers, while virtually synchronous group multicast is implemented by service end-points running at the clients. Specifically, the paper defines service semantics for the client-server interface, that is, for the group membership service. The paper then specifies virtually synchronous semantics for the new group multicast service, as a collection of commonly used safety and liveness properties. Finally, the paper presents new algorithms that use the defined group membership service to implement the specified properties. The algorithm that provides the complete virtually synchronous semantics executes in a single message round in parallel with the membership service's agreement on views, and is therefore more efficient than previously suggested algorithms providing such semantics.

1 Introduction

Group communication systems are powerful building blocks that facilitate the development of fault-tolerant distributed applications (see [1, 20, 5] for discussion of the utility of group communication systems). Group communication provides the notion of *group abstraction*, which allows processes to be easily organized in multicast groups. Group communication systems typically integrate two types of services: group membership and reliable group multicast. The membership service maintains a listing of the currently active and connected group members and delivers this information to its clients whenever it changes. The output of the membership service is called a *view*. Reliable multicast services that deliver messages to the current view members complement the membership service. In this paper, we present a novel group multicast service.

Group communication systems usually run in asynchronous fault-prone environments. In such environments, group communication systems generally provide some variant of *virtual synchrony* semantics which synchronize membership notifications with regular messages and thus simulate a "benign" world in which message delivery is reliable within the set of connected processes. Such semantics are especially useful for constructing fault-tolerant applications that maintain consistent replicated state of some sort (e.g., [7, 15]). The key aspect of virtual synchrony is the semantics of interleaving of message send and delivery events with view delivery events. In order to reason about this interleaving, we associate message send and delivery events with views: we say that an event $\mathtt{e}$ occurs at a process $\mathtt{p}$ *in view* $\mathtt{v}$ if $\mathtt{v}$ was the last view delivered to $\mathtt{p}$ before $\mathtt{e}$, or a default initial view $\mathtt{v}_\mathtt{p}$ if no such view was delivered.

Many variants of virtual synchrony semantics have been suggested (e.g., [17, 8, 20, 18, 7]). A key property specified by nearly all of these (e.g., [17, 8, 20, 18]) is that processes moving together from a view $\mathtt{v}$ to another view $\mathtt{v}'$ deliver the same messages in $\mathtt{v}$. This property allows applications to avoid costly re-synchronization following certain view changes. Our service specification includes this property, as well as several additional safety and liveness properties. We present our specifications in Section 4.

During the period in which the group communication service is attempting to reach agreement on a view, processes may attempt to join/re-join. In such cases, previously suggested virtual synchrony algorithms, e.g., [8, 18], can have the current invocation of the membership and virtual synchrony proceed to termination without adding the joining processes, and then immediately start an attempt to add them. This strategy results in overhead (e.g., increased network load) because applications react to such outdated views just as they do to any other view, e.g., by re-synchronizing with the new members. Moreover, this strategy precludes situations when applications may rely on virtual synchrony to avoid the costly re-synchronizations all together. For example, consider a transient failure when a process $\mathtt{p}$ is unsuspected right after an attempt to remove $\mathtt{p}$ from the membership has started. Existing algorithms typically deliver a view excluding $\mathtt{p}$ and then re-invoke the algorithm to allow $\mathtt{p}$ to re-join. Since $\mathtt{p}$ does not move into the resulting view from the same view as the rest of the processes, these processes cannot rely on the key property of virtual synchrony to avoid re-synchronizing with `p`. In contrast, our algorithm never delivers views that reflect a membership that is already known to be out of date.

Traditionally, virtual synchrony semantics were implemented by algorithms that were integrated with group membership algorithms (e.g., in [8, 9, 2]). In contrast, our group multicast service is designed for a client-server architecture in which a small set of dedicated membership servers maintains client membership information (i.e., which clients are members of which group). This architecture was designed to provide scalable membership services in wide area networks (cf. [3]). Our virtual synchrony algorithm acts as the client of an *external* membership service.

Introducing the client-server design poses a major challenge: One has to define an interface by which a membership server interacts with its clients, in a way that would allow for simple and efficient implementations of both group membership (by the membership servers), and virtual synchrony (by service end-points at the clients). Such an interface has to provide sufficient level of synchronization to allow the virtual synchrony algorithm to reach agreement upon the set of messages delivered in the old view in *parallel* with the servers' agreement on views. At the same time, the virtual synchrony algorithm should avoid imposing limitations on the membership's choice of views (as explained above). In addition, one has to try to minimize the communication overhead induced by the client-server interaction.

We have designed an interface that addresses the challenges above. Our interface consists of two types of messages sent from membership servers to their clients: When a server engages in a view change, it sends its clients a `start_change` message. Each `start_change` message contains a locally unique identifier. This identifier is *not* globally agreed upon: `start_change` messages sent to different processes can contain different identifiers. Once the server agrees upon the new view with the other servers, it sends a `view` message to its clients. The view contains information that maps clients to the last `start_change` identifiers they received before receiving this view. A similar view structure is suggested in [18], for the purpose of not having concurrent views intersect. The servers do not need to hear from their clients in order to complete the algorithm.

Our interface allows for straightforward and efficient implementations of both membership and virtual synchrony. The algorithm we present in Section 5 exploits this interface to achieve virtual synchrony in a single round. We have implemented this algorithm (in C++) using the scalable one-round membership algorithm of [11]. The virtual synchrony round and the membership round are conducted in parallel: once the end-points receive the `start_change` notifications, they send each other special synchronization messages which allow them to agree upon the set of messages to be delivered before moving to the new view. We are not aware of any other algorithm that implements virtual synchrony in one communication round without pre-agreement upon a globally unique identifier while also not imposing restrictions on the membership's choice of the next view.

Throughout this paper we use the I/O automaton formalism (cf. [16], Ch. 8) to provide rigorous specifications and algorithm descriptions. Previously suggested I/O automaton-style specifications of group communication systems (e.g., [7, 10]) used a single abstract automaton to represent multiple properties of the same system component and presented a single algorithm automaton that implements all of these properties. Thus, no means were provided for reasoning about a subset of the properties, and it was often difficult to follow which part of the algorithm implements which part of the specification. We address this shortcoming by specifying separate properties as separate abstract automata, and by incrementally constructing the algorithm that implements them – in each step adding support for an additional property – using a novel *inheritance*-based construct, recently introduced to the I/O automaton model [14, 13]. This paper informally argues the algorithm's correctness; a formal correctness proof by simulation is included in the full paper [12].

2 Formal model and notation

In the I/O automaton model (cf. [16], Ch. 8), a system component is described as a state-machine, called an *I/O automaton*. The transitions of this state-machine are associated with named actions, which are classified as either *input*, *output*, or *internal*. Input and output actions model the component's interaction with other components, while internal actions are externally-unobservable.

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

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

When reasoning about an automaton, we are only interested in its externally-observable behavior as reflected in its traces. There are two types of trace properties: *safety* and *liveness*. Safety properties usually specify that some particular bad thing never happens. In this paper we specify safety properties using 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. Liveness properties usually specify that some good thing eventually happens. An implementation automaton satisfies a liveness property if the property holds in all of its *fair* traces.

The *composition operation* defines how automata interact via their input and output actions: It matches output and input actions with the same name in different component automata; when a component automaton performs a step involving an output action, so do all components that have this action as an input one. When reasoning about a certain system component, we compose it with abstract specification automata that specify the behavior of its environment.

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

We use a novel *inheritance*-based formal concept, recently introduced into the I/O automaton model [14, 13]. 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:

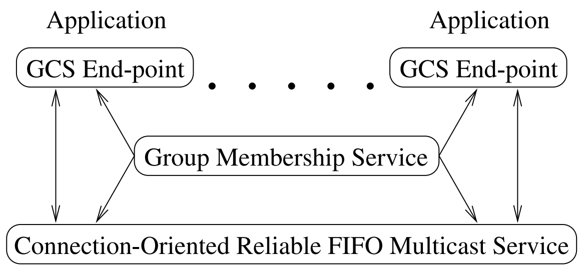

Fig. 1. The client-server architecture.

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, to ensure 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.

3 Environment specification

Our service is implemented in an asynchronous message-passing environment. Processes can crash, communication links may fail and may later recover, possibly causing network partitions and merges. In [12], we also model process recovery.

The service is implemented by *group communication service (GCS) end-points* running as clients of an external membership service whose specification appears in Section 3.1. The end-points communicate with each other using a reliable FIFO multicast service which we describe in Section 3.2, as depicted in Figure 1. We use the words "process" and "end-point" interchangeably.

3.1 The membership service

In Figure 2 we specify an external membership service whose interface consists of two output events:

`start_change`$_p$(`cid`, `set`) notifies process `p` that the membership service is attempting to form a new view with the members of `set`; `cid` is a local identifier.

`view`$_p$(`v`) notifies process `p` of the new 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_change identifiers. Two views are the same if they consist of identical triples.

The membership specification captures two basic membership properties, which are fulfilled by virtually all group membership services (e.g., [2, 8, 4, 11, 18]): *Self Inclusion* requires every view delivered to an end-point `p` to include `p` as a member, and *Local Monotonicity* requires that view identifiers delivered to `p` be monotonically increasing.

In addition, the MBRSHP automaton specifies, using the `mode`[`p`] variable, that the membership service must precede every view `v` sent to end-point `p` with at least one `start_change` notification to `p`. It also requires that, for every view `v` sent to `p`, the start_change identifier `v.startId`(`p`) be the same as the `cid` of the latest `start_change` issued to `p` before the view, and that `v.set` be a subset of the `set` suggested in that `start_change`. Note that this specification does allow the membership service to add new processes while it is reconfiguring, as long as a new `start_change` is sent to the clients. Also note that the specified service is *partitionable* [20, 4], i.e., allows several disjoint views to exist concurrently.

The specification allows for simple and efficient distributed implementations, e.g., [11], as well as many other existing membership algorithms (e.g., [2]) which could be easily extended to provide the specified interface and semantics. In a possible implementation, a small number of servers could support a large number of clients, communicating with them asynchronously via FIFO ordered channels. Fault-tolerant implementations that support client migration are also possible if the server name is included in the start_change identifier to guarantee its local uniqueness.

3.2 The reliable FIFO multicast service

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 (e.g., [9, 4]) implement virtual synchrony over similar underlying reliable communication substrates. In our implementation, we currently use the service of [19].

```
AUTOMATON MBRSHP

Type StartChangeId: Total-order; cid_0 is smallest.
    ViewId: Partial-order; vid_0 is smallest.
    View: ViewId x SetOf(Proc) x
                       x (Proc → StartChangeId)
Def  v_p  =  ⟨vid_0, {p}, {(p →cid_0)}⟩

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

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

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


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

Fig. 2. Membership service safety spec.

Figure 3 presents a centralized automaton CO_RFIFO which specifies a multicast service appropriate for our group communication algorithm. CO_RFIFO maintains a FIFO queue channel[p][q] for every pair of end-points. An input action $\texttt{send}_\texttt{p}$(set, m) models the multicast of message m from end-point p to the end-points listed in the set by appending m to the queues channel[p][q] for every end-point q in set.

The $\texttt{deliver}_{\texttt{p,q}}(\texttt{m})$ action removes the first message from $\texttt{channel}[\texttt{p}][\texttt{q}]$ and delivers it to $\texttt{q}$.

An end-point $\texttt{p}$ may use the action $\texttt{reliable}_{\texttt{p}}(\texttt{set})$ to require CO_RFIFO to maintain reliable (gap-free) FIFO connections to the end-points in $\texttt{set}$. For every process $\texttt{q}$ not in this $\texttt{set}$, CO_RFIFO may lose an arbitrary suffix of the messages sent from $\texttt{p}$ to $\texttt{q}$, as modeled by the action $\texttt{lose}(\texttt{p}, \texttt{q})$.

In specifying liveness of CO_RFIFO, we require that messages sent to live and connected processes eventually reach their destinations. We formulate this property by defining every $\texttt{deliver}_{\texttt{p,q}}(\texttt{m})$ to be a task if and only if $\texttt{q}$ is a member of $\texttt{live\_set}[\texttt{p}]$, a special variable periodically set by input actions $\texttt{live}_{\texttt{p}}(\texttt{set})$. The $\texttt{live}_{\texttt{p}}(\texttt{set})$ inputs are assumed to reflect the real state of the network, that is, the set of processes which are really alive and connected to $\texttt{p}$. Notice that we could not use the variable $\texttt{reliable\_set}[\texttt{p}]$ in this formulation because it is controlled by the client and thus does not necessarily reflect the *real* network situation.

4 GCS Specifications

We present the safety and liveness properties satisfied by our group communication service in Sections 4.1 and 4.2 respectively. These properties have been proven to be useful for many distributed applications (see [20]).

4.1 Safety properties

We present our safety specifications in four steps, as four automata: In Section 4.1 we specify a simple group communication service that provides reliable FIFO multicast within views. In Section 4.1 we extend the specification of Section 4.1 to also require that processes moving together from view $\texttt{v}$ to view $\texttt{v}'$ deliver the same messages in view $\texttt{v}$. In Section 4.1 we specify a service which provides *transitional sets* (first presented as part of *Extended Virtual Synchrony (EVS)* [17]). In Section 4.1 we specify the Self Delivery property which requires processes to deliver their own messages. The specified services are partitionable.

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

This specification captures the following properties:

- Views delivered to the application satisfy Local Monotonicity and Self Inclusion (cf. Sec. 3.1).
- Messages are delivered in the same view in which they were sent. This property is useful for many applications (cf. [8, 20]) and appears in several systems and specifications (e.g., see [1, 2, 17, 7]). 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 [20]).
- 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 (e.g., causally or totally ordered multicast).

Virtual synchrony In this section we specify a virtually synchronous reliable FIFO multicast service, VS_RFIFO, as a child of the presented above WV_RFIFO automaton. The VS_RFIFO specification consists of the code given in both Figures 4 and 5.

In addition to the properties inherited from WV_RFIFO, the VS_RFIFO specification also requires that processes moving together from view v to view $\texttt{v}'$ deliver the same set of messages in v. To enforce this property, the specification introduces internal actions $\texttt{set\_cut}(\texttt{v},\texttt{v}',\texttt{c})$ that non-deterministically fix the set of messages to be delivered in view v by every process that moves from v to $\texttt{v}'$. This set is represented by the index of the last message to be delivered in v from each sender. Note that a process that delivers messages beyond an already established cut is not allowed to move into the view associated with the cut.

This property is commonly provided (e.g., [17, 8, 20, 18, 10]) and is often called *Virtual Synchrony* by itself. It is especially useful for applications that implement data replication using the state machine approach (e.g., [7, 15]). Such applications may exploit Virtual Synchrony to avoid sending costly synchronization messages among processes that continue together from one view to the next.

```
AUTOMATON CO_RFIFO

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

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

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

  OUTPUT  deliver_{p,q}(m) hidden parameter live_set[p]
  pre: m = First(channel[p][q])
  eff: dequeue m from channel[p][q]


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

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

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

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

Fig. 3. CO_RFIFO service specification.

```
AUTOMATON WV_RFIFO : SPEC

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

State:
(∀ Proc p, View v) SeqOf(AppMsg) msgs[p][v],
                   initially empty
(∀ Proc p, Proc q) Int last_dlvrd[p][q], init. 0
(∀ Proc p) View current_view[p], init. 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. 4. WV_RFIFO service specification.

```
AUTOMATON VS_RFIFO : SPEC     MODIFIES  WV_RFIFO : SPEC

Signature Extension:
 Output:   view_p(v) modifies wv_rfifo.view_p(v)
 Internal: set_cut(v, v′, c), View v, View v′,
                          (Proc → Int)⊥  c new

State Extension:
 (∀ View v, v′) (Proc→Int)⊥  cut[v][v′], init. ⊥

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. VS_RFIFO service specification.

Transitional set While Virtual Synchrony is a useful property, a process that moves from view $\mathtt{v}$ to view $\mathtt{v}'$ cannot locally tell 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 Virtual Synchrony property, application processes need to be told which other processes move together with them from their old views in to their new views. The set of such processes is called a *transitional set*. The notion of a transitional set was first introduced as part of a special transitional view in the EVS [17] model. In our formulation (as in [20]), transitional sets are delivered to the applications together with (regular) views, as an additional parameter $\mathtt{T}$. The delivery of transitional sets satisfies the following property (cf. [20]):

Property 41 *The transitional set delivered by a process* $\mathtt{p}$ *when it moves from view* $\mathtt{v}$ *to view* $\mathtt{v}'$ *is a subset of* $\mathtt{v.set} \cap \mathtt{v}'\mathtt{.set}$ *that includes (a) all the processes (including* $\mathtt{p}$*) that move directly from* $\mathtt{v}$ *to* $\mathtt{v}'$ *and (b) no member of* $\mathtt{v}'\mathtt{.set}$ *that moves to* $\mathtt{v}'$ *from any view other than* $\mathtt{v}$*.*

Note that processes that move to the same view from different views deliver different transitional sets.

Figure 6 contains an automaton (without inheritance) specifying the Transitional Set property. Before $\mathtt{p}$ can move from view $\mathtt{v}$ to $\mathtt{v}'$, each member $\mathtt{q}$ of $\mathtt{v.set} \cap \mathtt{v}'\mathtt{.set}$ must execute $\mathtt{set\_prev\_view_q}(\mathtt{v}')$ to "declare" the view from which it intends to move to $\mathtt{v}'$; this action sets $\mathtt{prev\_view}[\mathtt{q}][\mathtt{v}']$ to $\mathtt{q}$'s current view. The transitional set delivered by $\mathtt{p}$ with $\mathtt{v}'$ is then computed to consist of those $\mathtt{q}$ in $\mathtt{v.set} \cap \mathtt{v}'\mathtt{.set}$ for which $\mathtt{prev\_view}[\mathtt{q}][\mathtt{v}'] = \mathtt{v}$.

Self delivery In Figure 7 we modify the WV_RFIFO specification automaton (Fig. 4) to capture the Self Delivery property by forbidding an end-point $\mathtt{p}$ to move from view $\mathtt{v}$ to $\mathtt{v}'$ before delivering all its own application messages sent in $\mathtt{v}$.

This *safety* property, when accompanied by the liveness property of Section 4.2, implies the Self Delivery *liveness* properties of [20] and [17], which require processes to *eventually* deliver their own messages.

4.2 Liveness property

In a fault-prone asynchronous model, it is not feasible to require that a group communication service be live in every execution. The only way to specify useful liveness properties without strengthening the communication model is to make these properties *conditional* on the underlying

```
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:
(∀  Proc p)  View current_view[p], initially v_p
(∀ Proc p, View v) View_⊥  prev_view[p][v], init. ⊥

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. 6. Transitional set specification.

```
AUTOMATON 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. 7. Self Delivery property specification.

network behavior (as specified, e.g., in [7, 20]). Since our GCS uses an external membership service, we condition its liveness on the behavior of the membership service (which itself is assumed to satisfy some meaningful liveness properties, e.g., those of [11]). Provided the membership eventually delivers the same view to all the view end-points and does not deliver any subsequent views (i.e., stabilizes), we require the end-points to eventually deliver this view and all the messages sent in this view to their applications.

Property 42 *Let* v *be a view with* $\texttt{v.set} = \texttt{S}$*. Let* α *be a fair execution of a group communication service* GCS *in which, for every* $\texttt{p} \in \texttt{S}$*,* MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$ *action occurs and is followed by neither* MBRSHP.$\texttt{view}_\texttt{p}$ *nor* MBRSHP.$\texttt{start\_change}_\texttt{p}$*. Then at each* $\texttt{p} \in \texttt{S}$*,* GCS.$\texttt{view}_\texttt{p}(\texttt{v})$ *eventually occurs. Furthermore, for every* GCS.$\texttt{send}_\texttt{p}(\texttt{m})$ *that occurs after* GCS.$\texttt{view}_\texttt{p}(\texttt{v})$*, and for every* $\texttt{q} \in \texttt{S}$*,* GCS.$\texttt{deliver}_\texttt{q}(\texttt{p}, \texttt{m})$ *also occurs.*

It is important to note that although our liveness property requires GCS to be live only in *certain* executions, any implementation which satisfies this property has to attempt to be live in *every* execution because of its inability to 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 GCS to reconfigure.

5 The group multicast algorithm

Our group communication service is implemented by a collection of GCS end-points, each running the same algorithm. Figure 8 (a) shows the interaction of a GCS end-point with its environment, MBRSHP and CO_RFIFO (see Sec. 3). 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 CO_RFIFO to send messages to other GCS end-points and to receive messages sent by other GCS end-points. When necessary, the end-point uses the action reliable 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_change and view notifications from the membership service.

The algorithm running at each end-point is constructed in steps, at each step adding support for a new property:

– First, we present an algorithm $\text{WV\_RFIFO}_\texttt{p}$ for an end-point of a within-view reliable FIFO multicast service.
– Then, in Section 5.2, we add support for the Virtual Synchrony and Transitional Set properties. We present a child $\text{VS\_RFIFO+TS}_\texttt{p}$ of $\text{WV\_RFIFO}_\texttt{p}$, and argue that the service built from $\text{VS\_RFIFO+TS}_\texttt{p}$ end-points satisfies safety specifications VSRFIFO : SPEC and TS : SPEC, and liveness Property 42.

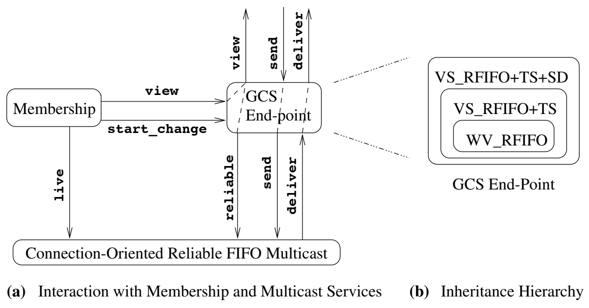

Fig. 8. GCS end-point and environment.

– Finally, in Section 5.3, we add support for Self Delivery. The resulting automaton VS_RFIFO+TS+SD$_p$ models a GCS end-point. Due to the use of inheritance, the service built from these end-points satisfies WV_RFIFO : SPEC, VSRFIFO : SPEC, and TS : SPEC. We argue that it also satisfies safety specification SELF : SPEC and liveness Property 42.

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 would eventually get executed.

When composing automata into a service, actions of the type MBRSHP.$\texttt{start\_change}_p(\texttt{id}, \texttt{set})$ are linked with CO_RFIFO.$\texttt{live}_p(\texttt{set})$, and MBRSHP.$\texttt{view}_p(\texttt{v})$ with CO_RFIFO.$\texttt{live}_p(\texttt{v.set})$. This way, the `live_set[p]` at CO_RFIFO matches the MBRSHP's perception of which processes are alive and connected to `p`. (We assume that every permanently disconnected end-point is eventually excluded by either a `start_change` or a `view` notification.) Also, in the composed system, all output actions except the application interface are reclassified as internal.

We present our algorithm at a level that would be easy to follow and then supplement this presentation with a discussion of some important, practical optimizations.

5.1 Within-view reliable FIFO multicast algorithm

In this section we present algorithm WV_RFIFO$_p$ for an end-point `p` of a service that interacts with MBRSHP and CO_RFIFO services and satisfies the WV_RFIFO : SPEC safety specification and liveness Property 42.

The MBRSHP and CO_RFIFO services by themselves already provide most of the properties required by the WV_RFIFO : SPEC specification:

```
AUTOMATON WV_RFIFO_p

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

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

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

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

  SetOf(Proc) reliable_set, initially v_p.set

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

 OUTPUT  view_p(v)
 pre: v = mbrshp_view ≠ 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
 eff: reliable_set ← set

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

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

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

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

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

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

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

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

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

MBRSHP generates views that satisfy Local Monotonicity and Self Inclusion, and CO_RFIFO provides gap-free FIFO communication. Since WV_RFIFO$_p$ can just forward to its application the views generated by MBRSHP and can use CO_RFIFO to multicast application messages to other end-points, it only needs to ensure that messages are delivered in the same views in which they were sent. This can be done simply by tagging messages with the views in which they were sent and by allowing delivery of a message when its view tag matches the end-point's current view.

As an optimization of this idea, instead of tagging each message with a view in which it was sent, our algorithm sends a single, special view_msg(v) to all members of view v before sending them application messages in that view. An end-point can deduce the view in which an application message is sent from the latest view_msg(v) received from the application message sender.

The algorithm is captured in the WV_RFIFO$_\mathtt{p}$ automaton of Figure 9. Note that, instead of blindly relying on CO_RFIFO regarding which messages get delivered in a given view, WV_RFIFO allows processes to forward application messages on behalf of other processes. The code of WV_RFIFO$_\mathtt{p}$ does not specify a particular forwarding strategy – it allows for non-deterministic forwarding of messages. Without this, more refined versions and extensions of WV_RFIFO would not be able to introduce a specific forwarding strategy (as we do in VS_RFIFO+TS by adding a precondition on the action that sends forwarded messages).

There is also another place where the code leaves a non-deterministic choice: it is in handling of the `reliable_set` of CO_RFIFO. The code allows it to be an arbitrary superset of `current_view.set`. This set is further restricted in a child VS_RFIFO+TS$_\mathtt{p}$ of WV_RFIFO$_\mathtt{p}$.

The correctness of WV_RFIFO follows from the use of ordered message queues, the safety and liveness properties of CO_RFIFO, and the safety properties of MBRSHP. A formal proof is given in [12].

Also note that the presented code never removes messages from its buffers. An actual implementation can and should employ some sort of a garbage collection mechanism, for example discard messages sent in older views when moving in to a new view.

5.2 Virtual synchrony and transitional sets

The WV_RFIFO service presented above guarantees that in each view `v` every member delivers some prefix of the FIFO ordered messages sent by each end-point in `v`. The VS_RFIFO+TS service presented in this section extends WV_RFIFO to also guarantee that those end-points which transition directly from view `v` to the same view $\mathtt{v}'$ deliver not just "some" prefixes but "the same" prefixes of the FIFO ordered messages sent by each end-point in view `v` (cf. Sec. 4.1). Moreover, every view delivery is accompanied by a transitional set `T` that satisfies the Transitional Set property of Sec. 4.1.

In order to satisfy these two properties, an end-point moving from a view `v` to a view $\mathtt{v}'$ must first learn which other end-points may transition from `v` to $\mathtt{v}'$ and must agree with them on the lengths of the prefixes they need to deliver. In a nutshell, here is how the VS_RFIFO+TS service accomplishes this: Each time an end-point `p` is notified via MBRSHP.$\mathtt{start\_change}_\mathtt{p}(\mathtt{cid}, \mathtt{set})$ of the MBRSHP's attempt to form a new view, `p` reliably sends to `set` a synchronization message tagged with `cid`. When MBRSHP.$\mathtt{view}_\mathtt{p}(\mathtt{v}')$ is delivered to `p`, `p` uses the $\mathtt{v}'$`.startId` mapping to determine which synchronization message to use from each end-point `q` in $\mathtt{v.set} \cap \mathtt{v}'\mathtt{.set}$; it uses the one tagged with `v'.startId(q)`. As a result, all end-points that move from view `v` to `v'` use the same set of synchronization messages for computing the transitional set and the set of messages to be delivered to their application clients before `v'`. Notice that, by enriching views with the `startId` mapping, we eliminate the need to pre-agree on a common tag for identifying which synchronization messages to consider for a given view.

```
AUTOMATON VS_RFIFO+TS_p   MODIFIES   WV_RFIFO_p

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

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

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

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

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

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

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

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

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

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

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

Algorithm details and safety argument Figure 10 presents the VS_RFIFO+TS$_p$ automaton as a child of WV_RFIFO$_p$. While there are no view changes, VS_RFIFO+TS$_p$ does not modify the behavior of WV_RFIFO$_p$. During a view change, VS_RFIFO+TS$_p$ sends and handles synchronization messages, and also restricts the delivery of application messages according to the synchronization messages associated with the new view.

Upon receiving a start_change$_p$(cid, set) notification from MBRSHP, end-point p stores ⟨cid, set⟩ in the variable start_change, tells CO_RFIFO to maintain reliable communication to the end-points in current_view $\cup$ set, and then sends a synchronization message tagged with cid to every end-point in set. The synchronization message contains p's current view v and a cut, which is a mapping from processes to indices; cut(q) is the index of the last message from q that p *commits* to deliver before delivering any view v′ with v′.startId(p) = cid.

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

End-point p is guaranteed to eventually receive all the application messages sent by the members of its transitional set T. However, p may fail to receive some of the application messages sent by disconnected end-points (not in T) although certain cuts of members of T commit to deliver these messages. Such messages need to be forwarded to p by the members of T that have them. These members of T deduce from the p's cut that p lacks these messages and use a ForwardingStrategyPredicate to compute which of them have to forward which missing messages to p. We describe some of the many possible such predicates in [12].

Liveness of VS_RFIFO+TS We show that, in a fair execution of VS_RFIFO+TS in which the same view v' is delivered to all its members as their last MBRSHP event, the three preconditions on the $\mathtt{view}_p(v', T_p)$ delivery are eventually satisfied for every $p \in v'.\mathtt{set}$:

1. Condition $v'.\mathtt{startId}(p) = \mathtt{start\_change.id}$ remains true since by the assumption there are no subsequent `start_change` events at `p`.
2. End-point `p` eventually receives synchronization messages tagged with the "right" `cid` from everybody in $\mathtt{v.set} \cap \mathtt{v'.set}$ 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 in the cuts of the members of the transitional set T_p because for each such message there is at least one end-point in T_p that has the message in its `msgs` buffer and that would reliably forward it to `p` (according to the `ForwardingStrategyPredicate`) if so necessary. Also, `p` never delivers any messages beyond those committed to in the cuts of the members of T_p because of the precondition on application message delivery.

Optimizations Notice that end-point `p` does not need to send its current view and its cut to end-points which are not in `current_view.set` because `p` cannot be included in their transitional sets. Nevertheless, these end-points may wait to hear from `p` as `p` may still be in their current views. Therefore, in our algorithm, `p` sends synchronization messages to all the end-points in `start_change.set`. As an optimization, `p` could send a smaller synchronization message to processes in $\mathtt{start\_change.set} - \mathtt{current\_view.set}$, containing its `start_change.id` only (but neither a view nor a cut). The recipients of this message would know not to include `p` in their transitional sets for views v' with $v'.\mathtt{startId}(p) =$ `p`'s `start_change.id`. When using this optimization, `p` also does not need to include its current view in the synchronization messages sent to $\mathtt{current\_view.set} - \mathtt{start\_change.set}$, since the view information can be deduced from `p`'s `view_msg`.

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

5.3 Self delivery

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

In order to implement Self Delivery and Virtual Synchrony together in a live manner, each end-point must *block* its application from sending new messages while a view change is taking place (as proven in [8]). Therefore, we modify VS_RFIFO+TS$_p$ to have an output action `block` and an input action `block_ok`, and we assume that the application at end-point `p` has the matching actions and that it eventually responds to every `block` request with a `block_ok` response and subsequently refrains from sending messages until a `view` is delivered to it. In [12], we model this assumption with an abstract application automaton.

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

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

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

Signature Extension:
Input:  block_ok_p() new

Output: block_p() new
        view_p(v,T) modifies vs_rfifo+ts.view_p(v,T)
        co_rfifo.send_p(set, m) modifies
        vs_rfifo+ts.co_rfifo.send_p(set,SyncMsg m)

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

Transition Restriction:
OUTPUT  block_p()
pre: start_change ≠ ⊥
     block_status = unblocked
eff: block_status ← requested

INPUT  block_ok_p()
eff: block_status ← blocked

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

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

Fig. 11. $\mathtt{GCS_p}$ end-point automaton.

6 Conclusions

We have constructed a virtually synchronous group multicast algorithm which exchanges one round of synchronization messages during reconfiguration, in parallel with the execution of a group membership algorithm. In contrast to previously suggested virtual synchrony algorithms, our algorithm does not require processes to conduct an additional communication round in order to pre-agree upon a globally unique identifier and does not impose restrictions on membership service's choice of views. We are not aware of any other algorithm that has both of these features.

These features are achieved by virtue of a simple yet powerful idea: Membership service issues a *locally* unique start-change identifier every time it has new information about client membership. The inclusion of such identifiers in views eliminates the need to tag clients' messages with a common (globally unique) identifier.

The start-change interface is an important aspect of the design of a client-server oriented group communication service which decouples membership maintenance from group multicast in order to provide scalable group membership services in WANs. Maestro [6] and the service of [18] also separate the maintenance of membership from group multicast. Unlike Maestro [6], in our design, the client does not wait for the membership to agree upon a globally unique identifier before starting the virtual synchrony algorithm, and the membership service does not wait for responses from clients asserting that virtual synchrony was achieved before delivering views. Unlike [18], our service does not impose restrictions on the membership service's choice of views, thereby allowing its applications to benefit from Virtual Synchrony in more cases (as explained in Introduction).

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

Our service is implemented as part of a novel architecture for scalable group communication in WANs. After testing its current scalability limits, we intend to explore ways to improve the scallability further by incorporating a two-tier hierarchy into our algorithm, as suggested by Guo et al. [9]. With this approach, processes would send synchronization messages to their designated leaders, who would in turn exchange only the cumulative information among themselves. The framework in which we presented our algorithm allows us to incorporate extensions such as this one.

In [12] we formally prove the correctness of our algorithms. In particular, we prove the safety properties by defining simulation relations from the algorithm automata to the specification automata. The incremental way in which we have constructed our algorithms and specifications allows us to also construct the simulation proof incrementally. For example, in order to prove that VS_RFIFO+TS simulates VS_RFIFO+TS : SPEC we extend the simulation relation from WV_RFIFO to WV_RFIFO : SPEC and reason solely about the extension, without repeating the reasoning about the parent components. This reuse is justified by the Proof Extension theorem of [14, 13].

6.1 Acknowledgments

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

References

1. ACM. *Communications of the ACM 39(4), special issue on Group Communications Systems*, April 1996.
2. Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4), November 1995.
3. T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In Marios Mavronicolas, Michael Merritt, and Nir Shavit, editors, *Networks in Distributed Computing (DIMACS workshop)*, volume 45 of *DIMACS*, pages 23–42. American Mathematical Society, 1998.
4. Ö. Babaoğlu, R. Davoli, and A. Montresor. Partitionalbe group membership: Specification and algorithms. TR UBLCS97-1, Department of Computer Science, University of Bologna, January 1997.
5. K. Birman. *Building Secure and Reliable Network Applications*. Manning, 1996.
6. K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware support for distributed multimedia and collaborative computing. In *Multimedia Computing and Networking (MMCN98)*, 1998.
7. A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001. Previous version appeared in PODC 1997.
8. Roy Friedman and Robbert van Renesse. Strong and Weak Virtual Synchrony in Horus. TR 95-1537, dept. of Computer Science, Cornell University, August 1995.
9. Katherine Guo, Werner Vogels, and Robbert van Renesse. Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication. In *7th ACM SIGOPS European Workshop*, September 1996.
10. Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for ensemble layers. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer-Verlag, March 1999.
11. I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. In *20th International Conference on Distributed Computing Systems (ICDCS)*, pages 356–365, April 2000. Full version: MIT Technical Memorandum MIT-LCS-TM-593a, June 1999, revised September 2000.

12. Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications, algorithms, and proofs. Technical Report MIT-LCS-TR-794, MIT Laboratory for Computer Science, November 1999.
13. Idit Keidar, Roger Khazan, Nancy Lynch, and Alex Shvartsman. An inheritance-based technique for building simulation proofs incrementally. To appear. Previous version in ICSE 2000, pp. 478–487.
14. Idit Keidar, Roger Khazan, Nancy Lynch, and Alex Shvartsman. An inheritance-based technique for building simulation proofs incrementally. In *22nd International Conference on Software Engineering (ICSE)*, pages 478–487, June 2000.
15. Roger Khazan, Alan Fekete, and Nancy Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on DIStributed Computing (DISC)*, pages 258–272, Andros, Greece, September 1998.
16. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
17. L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994. Full version: technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.
18. A. Schiper and A.M. Ricciardi. Virtually synchronous communication based on a weak failure suspector. In *23rd IEEE Fault-Tolerant Computing Symposium (FTCS)*, pages 534–543, June 1993.
19. Ilya Shnaiderman. Implementation of Reliable Datagram Service in the LAN environment. Lab project, The Hebrew University of Jerusalem, January 1999. http://www.cs.huji.ac.il/∼transis/publications.html.
20. R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report CS99-31, Institute of Computer Science, Hebrew University, Jerusalem, Israel, September 1999. Also Technical Report MIT-LCS-TR-790, Massachusetts Institute of Technology, Laboratory for Computer Science and Technical Report CS0964, Computer Science Department, the Technion, Haifa, Israel.