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 $e$ occurs at a process $p$ in view $v$ if $v$ was the last view delivered to $p$ before $e$, or a default initial view $v_p$ 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 $v$ to another view $v'$ deliver the same messages in $v$. 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 $p$ is unsuspected right after an attempt to remove $p$ from the membership has started. Existing algorithms typically deliver a view excluding $p$ and then re-invoke the algorithm to allow $p$ to re-join. Since $p$ 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++) us-
ing the scalable one-round membership algorithm of [11]. The virtual
synchrony round and the membership round are conducted in parallel:
onece the end-points receive the \texttt{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 com-
munication systems (e.g., [7,10]) used a single abstract automaton to
represent multiple properties of the same system component and pre-
sented a single algorithm automaton that implements all of these prop-
erties. 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 au-
tomata, and by incrementally constructing the algorithm that imple-
ments them – in each step adding support for an additional property
– using a novel \textit{inheritance}-based construct, recently introduced to the
I/O automaton model [14,13]. This paper informally argues the algo-
rithm’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 ac-
tions 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 \( \pi \) is said to be \textit{enabled} in a state \( s \) if the automaton has
a transition of the form \((s, \pi, s')\); input actions are enabled in every
state. An \textit{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 com-
ponents 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 \( S \bot \) refers to the set \( S \cup \{ \bot \} \)). Transitions are grouped by
action name, and are specified as a list of triples consisting of an action
name (possibly with parameters), a \( \text{pre} : \) block with preconditions on
the states in which the action is enabled, and an \( \text{eff} : \) block which
specifies how the pre-state is modified atomically to yield the post-
state.

We use a novel inheritance-based formal concept, recently intro-
duced into the I/O automaton model \([14,13]\). A child automaton is
specified as a modification of the parent automaton’s code. When pre-
senting 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 \texttt{parent.action(parameters)}. We next specify the \textit{state extension} consisting of new state variables added by the child. Finally, we describe the \textit{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.

\section{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 \textit{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.

\subsection{The membership service}

In Figure 2 we specify an external membership service whose interface consists of two output events:
Start-change \( (cid, set) \) 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 \( v \) 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 variable, that the membership service must precede every view 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 MBRSHIP

Type StartChangeId: Total-order; cid0 is smallest.
ViewId: Partial-order; vid0 is smallest.
View: ViewId x SetOf(Proc) x
   (Proc → StartChangeId)

Def \( v_p = (vid_0, \{p\}, \{(p \rightarrow cid_0)\}) \)

Signature:
Output: start\_change\_p(cid, set), Proc p,
       StartChangeId cid, SetOf(Proc) set
       view\_p(v), Proc p, View v

State:
(\( \forall \) Proc p) View mbrshp\_view[p], initially \( v_p \)
(\( \forall \) Proc p) (StartChangeId x SetOf(Proc))
   start\_change[p], initially \( (cid_0, \{\}) \)
(\( \forall \) Proc p) mode[p] \( \in \) {normal, change\_started},
   initially normal

Transitions:
OUTPUT start\_change\_p(cid, set)
pre: cid > start\_change[p].id
   p \( \in \) set
eff: start\_change[p] \( \leftarrow \) (cid, set)
   mode[p] \( \leftarrow \) change\_started

OUTPUT view\_p(v)
pre: p \( \in \) v.set \( \land \) v.id > mbrshp\_view[p].id
   v.set \( \subseteq \) start\_change[p].set
   v.startId(p) = start\_change[p].id
   mode[p] = change\_started
eff: mbrshp\_view[p] \( \leftarrow \) v
   mode[p] \( \leftarrow \) 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 send\_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 $\text{deliver}_{p,q}(m)$ action removes the first message from $\text{channel}_{p|q}$ and delivers it to $q$.

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

In specifying liveness of $\text{CO}_RFIFO$, we require that messages sent to live and connected processes eventually reach their destinations. We formulate this property by defining every $\text{deliver}_{p,q}(m)$ to be a task if and only if $q$ is a member of $\text{live}_{\text{set}}[p]$, a special variable periodically set by input actions $\text{live}_p(\text{set})$. The $\text{live}_p(\text{set})$ inputs are assumed to reflect the real state of the network, that is, the set of processes which are really alive and connected to $p$. Notice that we could not use the variable $\text{reliable}_{\text{set}}[p]$ 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 $v$ to view $v'$ deliver the same messages in view $v$. In Section 4.1 we specify a service which provides transitional sets (first presented as part of $\text{Extended Virtual Synchrony (EVS)}$ [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 ($\text{wv}_RFIFO$) service specification. The specification uses centralized queues $\text{msgs}[p|v]$ of application messages for each sender $p$ and view $v$. The action $\text{send}_{p}(m)$ models the multicast
of message $m$ from process $p$ to the members of $p$'s current view by appending $m$ to $\text{msgs}[p][\text{current\_view}[p]]$. The $\text{delivery}_{p}(q, m)$ action models the delivery to process $p$ of message $m$ sent by process $q$ while in $p$'s current view. The specification enforces gap-free FIFO ordered delivery of messages by using the variable $\text{last\_dlvrd}[q][p]$ 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, $\text{VS\_RFIFO}$, as a child of the presented above $\text{WV\_RFIFO}$ automaton. The $\text{VS\_RFIFO}$ specification consists of the code given in both Figures 4 and 5.

In addition to the properties inherited from $\text{WV\_RFIFO}$, the $\text{VS\_RFIFO}$ specification also requires that processes moving together from view $v$ to view $v'$ deliver the same set of messages in $v$. To enforce this property, the specification introduces internal actions $\text{set\_cut}(v, v', c)$ that non-deterministically fix the set of messages to be delivered in view $v$ by every process that moves from $v$ to $v'$. 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 CO₉RFIFO**

**Signature:**

Input:
- \( \text{send}_p(set, m) \), Proc \( p \), SetOf(Proc) \( set \), Msg \( m \)
- \( \text{reliable}_p(set) \), Proc \( p \), SetOf(Proc) \( set \)
- \( \text{live}_p(set) \), Proc \( p \), SetOf(Proc) \( set \)

Output: \( \text{deliver}_p,q(m) \), Proc \( p \), Proc \( q \), Msg \( m \)

Internal: \( \text{lose}(p,q) \), Proc \( p \), Proc \( q \)

**State:**

\( (\forall \text{ Proc } p, \text{ Proc } q) \) SeqOf(Msg) \( \text{channel}[p][q] \),

\( \text{initially empty} \)

\( (\forall \text{ Proc } p) \) SetOf(Proc) \( \text{reliable}_\text{set}[p] \), init. \( \{p\} \)

\( (\forall \text{ Proc } p) \) SetOf(Proc) \( \text{live}_\text{set}[p] \), init. \( \{p\} \)

**Transitions:**

**INPUT** \( \text{send}_p(set, m) \)

\( \text{eff}: (\forall q \in set) \) append \( m \) to \( \text{channel}[p][q] \)

**OUTPUT** \( \text{deliver}_p,q(m) \) hidden parameter \( \text{live}_\text{set}[p] \)

\( \text{pre}: m = \text{First}(\text{channel}[p][q]) \)

\( \text{eff}: \) dequeue \( m \) from \( \text{channel}[p][q] \)

**INPUT** \( \text{reliable}_p(set) \)

\( \text{eff}: \) \( \text{reliable}_\text{set}[p] \leftarrow set \)

**INTERNAL** \( \text{lose}(p, q) \)

\( \text{pre}: q \not\in \text{reliable}_\text{set}[p] \)

\( \text{eff}: \) dequeue last message from \( \text{channel}[p][q] \)

**INPUT** \( \text{live}_p(set) \)

\( \text{eff}: \) \( \text{live}_\text{set}[p] \leftarrow set \)

**Tasks:**

1. \( (\forall p)(\forall q \in \text{live}_\text{set}[p]) \) \( \{\text{deliver}_p,q(m)\} \)
2. \{\text{dummy()}\} \cup \{\text{deliver}_p,q(m) \mid q \not\in \text{live}_\text{set}[p]\}

\( \cup \{\text{lose}(p,q)\} \)

Fig. 3. CO₉RFIFO service specification.
**Fig. 4.** *wv_rfifo* service specification.
Signature Extension:
Output: \( \text{view}_p(v) \) modifies \( \text{wv}_\text{rfifo}.\text{view}_p(v) \)
Internal: set\_cut\( (v, v', c), \text{View} v, \text{View} v' \),
\( (\text{Proc} \rightarrow \text{Int}) \perp c \) new

State Extension:
\( (\forall \text{View} v, v') (\text{Proc} \rightarrow \text{Int}) \perp \text{cut}[v][v'], \text{init.} \perp \)

Transition Restriction:
OUTPUT \( \text{view}_p(v) \)
pre: cut[\text{current\_view}[p]] [v] \neq \perp
\( (\forall q) \text{last\_dlvrd}[q][p] = \text{cut}[\text{current\_view}[p]][v](q) \)

INTERNAL set\_cut\( (v, v', c) \)
pre: cut[v][v'] = \perp
eff: cut[v][v'] \leftarrow c

Fig. 5. \textsc{vs\_rfifo} service specification.
Transitional set While Virtual Synchrony is a useful property, a process that moves from view $v$ to view $v'$ cannot locally tell which of the processes in $v.set \cap v'.set$ move to view $v'$ directly from view $v$, and which move to $v'$ from some other view. In order for the application to be able to exploit the Virtual Synchrony property, application processes need to be told which other processes move together with them from their old views in to their new views. The set of such processes is called a transitional set. The notion of a transitional set was first introduced as part of a special transitional view in the EVS [17] model. In our formulation (as in [20]), transitional sets are delivered to the applications together with (regular) views, as an additional parameter $T$. The delivery of transitional sets satisfies the following property (cf. [20]):

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

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

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 $p$ to move from view $v$ to $v'$ before delivering all its own application messages sent in $v$.

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
Virtually Synchronous Group Multicast

**AUTOMATON TRANS_SET : SPEC**

**Signature:**
Output: $\text{view}_p(v, T)$, Proc p, View v, SetOf(Proc) T
Internal: $\text{set}\_\text{prev}\_\text{view}_p(v)$, Proc p, View v

**State:**
$(\forall \text{ Proc } p) \text{ View } \text{current}\_\text{view}[p]$, initially $v_p$
$(\forall \text{ Proc } p, \text{ View } v) \text{ View } \bot \text{ prev}\_\text{view}[p][v]$, init. $\bot$

**Transitions:**

**OUTPUT** $\text{view}_p(v, T)$
pre: $\text{prev}\_\text{view}[p][v] = \text{current}\_\text{view}[p]$
$(\forall q \in v.\text{set} \cap \text{current}\_\text{view}[p].\text{set})$
$\text{prev}\_\text{view}[q][v] \neq \bot$
$T = \{q \in v.\text{set} \cap \text{current}\_\text{view}[p].\text{set} | \text{prev}\_\text{view}[q][v] = \text{current}\_\text{view}[p]\}$
eff: $\text{current}\_\text{view}[p] \leftarrow v$

**INTERNAL** $\text{set}\_\text{prev}\_\text{view}_p(v)$
pre: $p \in v.\text{set}$
eff: $\text{prev}\_\text{view}[p][v] \leftarrow \text{current}\_\text{view}[p]$

*Fig. 6.* Transitional set specification.

**AUTOMATON SELF : SPEC** MODIFIES $\text{wv}_\text{rfifo : SPEC}$

**Signature Extension:**
Output: $\text{view}_p(v)$ modifies $\text{wv}_\text{rfifo}.\text{view}_p(v)$

**Transition Restriction:**

**OUTPUT** $\text{view}_p(v)$
pre: $\text{last}\_\text{dlvrd}[p][p]$
$= \text{LastIndexOf}(\text{msgs}[p][\text{current}\_\text{view}[p]])$

eff: $\text{current}\_\text{view}[p]$

*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 \( v.set = S \). Let \( \alpha \) be a fair execution of a group communication service \( GCS \) in which, for every \( p \in S \), \( \text{MBRSHP}.\text{view}_p(v) \) action occurs and is followed by neither \( \text{MBRSHP}.\text{view}_p \) nor \( \text{MBRSHP}.\text{start}_p \). Then at each \( p \in S \), \( GCS.\text{view}_p(v) \) eventually occurs. Furthermore, for every \( GCS.\text{send}_p(m) \) that occurs after \( GCS.\text{view}_p(v) \), and for every \( q \in S \), \( GCS.\text{deliver}_q(p,m) \) 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, \( \text{MBRSHP} \) and \( \text{co РФIFO} \) (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 \( \text{co РФIFO} \) 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 \text{reliable} to inform \( \text{co РФIFO} \) of the set of end-points to which \( \text{co РФIFO} \) must maintain reliable (gap-free) FIFO connections. The \( GCS \) end-point also receives \( \text{start}_p \) and \( \text{view}_p \) 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 \( \text{wv РФIFO}_p \) 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 \( \text{vs РФIFO} + \text{TS}_p \) of \( \text{wv РФIFO}_p \), and argue that the service built from \( \text{vs РФIFO} + \text{TS}_p \) end-points satisfies safety specifications \( \text{VS РФIFO : SPEC} \) and \( \text{TS : SPEC} \), and liveness Property 42.
Finally, in Section 5.3, we add support for Self Delivery. The resulting automaton VS_RFIFO+TS+SD models a GCS end-point. Due to the use of inheritance, the service built from these end-points satisfies WV_RFIFO : SPEC, VS_RFIFO : 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 MBRSHIP.start_change(id, set) are linked with CO_RFIFO.live_p(set), and MBRSHIP.view_p(v) with CO_RFIFO.live_p(v.set). This way, the live set[p] at CO_RFIFO matches the MBRSHIP’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 MBRSHIP and CO_RFIFO services and satisfies the WV_RFIFO : SPEC safety specification and liveness Property 42.

The MBRSHIP and CO_RFIFO services by themselves already provide most of the properties required by the WV_RFIFO : SPEC specification:
MBRSHP generates views that satisfy Local Monotonicity and Self Inclusion, and CO_RFIFO provides gap-free FIFO communication. Since WV_RFIFO 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 \texttt{WVRFIFO}\textsubscript{p} automaton of Figure 9. Note that, instead of blindly relying on \texttt{COR FIFO} regarding which messages get delivered in a given view, \texttt{WVR FIFO} allows processes to forward application messages on behalf of other processes. The code of \texttt{WVR FIFO}\textsubscript{p} does not specify a particular forwarding strategy – it allows for non-deterministic forwarding of messages. Without this, more refined versions and extensions of \texttt{WVR FIFO} would not be able to introduce a specific forwarding strategy (as we do in \texttt{VR FIFO}+\texttt{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 \texttt{reliable set} of \texttt{COR FIFO}. The code allows it to be an arbitrary superset of \texttt{current view set}. This set is further restricted in a child \texttt{VR FIFO}+\texttt{TS}\textsubscript{p} of \texttt{WVR FIFO}\textsubscript{p}.

The correctness of \texttt{WVR FIFO} follows from the use of ordered message queues, the safety and liveness properties of \texttt{COR FIFO}, and the safety properties of \texttt{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 into a new view.

### 5.2 Virtual synchrony and transitional sets

The \texttt{WVR FIFO} service presented above guarantees that in each view \texttt{v} every member delivers some prefix of the FIFO ordered messages sent by each end-point in \texttt{v}. The \texttt{VR FIFO}+\texttt{TS} service presented in this section extends \texttt{WVR FIFO} to also guarantee that those end-points which transition directly from view \texttt{v} to the same view \texttt{v'} deliver not just “some” prefixes but “the same” prefixes of the FIFO ordered messages sent by each end-point in view \texttt{v} (cf. Sec. 4.1). Moreover, every view delivery is accompanied by a transitional set \texttt{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 \texttt{v} to a view \texttt{v'} must first learn which other end-points may transition from \texttt{v} to \texttt{v'} and must agree with them on the lengths of the prefixes they need to deliver. In a nutshell, here is how the \texttt{VR FIFO}+\texttt{TS} service accomplishes this: Each time an end-point \texttt{p} is notified via \texttt{MBRSHP.start change}(\texttt{cid, set}) of the \texttt{MBRSHP}’s attempt to form a new view, \texttt{p} reliably sends to \texttt{set} a synchronization message tagged with \texttt{cid}. When \texttt{MBRSHP.view}(\texttt{v'}) is delivered to \texttt{p}, \texttt{p} uses the \texttt{v'.startId} mapping to determine which synchronization message to use from each end-point \texttt{q} in \texttt{v.set} \(\cap\) \texttt{v'.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.

**Fig. 10.** Virtually synchronous reliable FIFO multicast and transitional set end-point automaton.
Algorithm details and safety argument  Figure 10 presents the \textit{vs$_{\text{RFIFO+TS}}_p$} automaton as a child of \textit{wv$_{\text{RFIFO}}_p$}. While there are no view changes, \textit{vs$_{\text{RFIFO+TS}}_p$} does not modify the behavior of \textit{wv$_{\text{RFIFO}}_p$}. During a view change, \textit{vs$_{\text{RFIFO+TS}}_p$} 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 \textit{start\_change$_p$(cid, set)} notification from \texttt{MBRSHP}, end-point \texttt{p} stores \texttt{(cid, set)} in the variable \texttt{start\_change}, tells \texttt{co\_RFIFO} to maintain reliable communication to the end-points in \texttt{current\_view $\cup$ set}, and then sends a synchronization message tagged with \texttt{cid} to every end-point in \texttt{set}. The synchronization message contains \texttt{p}'s current view \texttt{v} and a \texttt{cut}, which is a mapping from processes to indices; \texttt{cut(q)} is the index of the last message from \texttt{q} that \texttt{p} commits to deliver before delivering any view \texttt{v'} with \texttt{v'.startId(p) = cid}.

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

End-point \texttt{p} is guaranteed to eventually receive all the application messages sent by the members of its transitional set \texttt{T}. However, \texttt{p} may fail to receive some of the application messages sent by disconnected end-points (not in \texttt{T}) although certain cuts of members of \texttt{T} commit to deliver these messages. Such messages need to be forwarded to \texttt{p} by the members of \texttt{T} that have them. These members of \texttt{T} deduce from the \texttt{p}'s cut that \texttt{p} lacks these messages and use a \texttt{ForwardingStrategyPredicate} to compute which of them have to forward which missing messages to \texttt{p}. We describe some of the many possible such predicates in \cite{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 $\text{view}_p(v', T_p)$ delivery are eventually satisfied for every $p \in v'.\text{set}$:

1. Condition $v'.\text{startId}(p) = \text{start_change}.id$ remains true since by the assumption there are no subsequent $\text{start_change}$ events at $p$.
2. End-point $p$ eventually receives synchronization messages tagged with the “right” cid from everybody in $v.\text{set} \cap v'.\text{set}$ 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 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 $\text{msgs}$ 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 $\text{current_view}.\text{set}$ 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 $\text{start_change}.\text{set}$. As an optimization, $p$ could send a smaller synchronization message to processes in $\text{start_change}.\text{set} - \text{current_view}.\text{set}$, containing its $\text{start_change}.id$ 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 $v'.\text{startId}(p) = p$'s $\text{start_change}.id$. When using this optimization, $p$ also does not need to include its current view in the synchronization messages sent to $\text{current_view}.\text{set} - \text{start_change}.\text{set}$, since the view information can be deduced from $p$’s $\text{view}_p$.

Another optimization can be used to minimize synchronization message sizes if we strengthen the membership specification to require a MBRSHP.$\text{start_change}$ to be sent every time the membership changes its mind about the next view. In this case, the latest MBRSHP.$\text{start_change}$ has the same membership as the delivered MBRSHP.$\text{view}$, and therefore the synchronization messages do not need
to include information about messages delivered from end-points in \texttt{start\_change.set} \cap \texttt{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, \texttt{gcs\_p}, we add support for Self Delivery to the \texttt{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 \texttt{block} its application from sending new messages while a view change is taking place (as proven in [8]). Therefore, we modify \texttt{vs\_rfifo\_ts\_p} to have an output action \texttt{block} and an input action \texttt{block\_ok}, and we assume that the application at end-point \texttt{p} has the matching actions and that it eventually responds to every \texttt{block} request with a \texttt{block\_ok} response and subsequently refrains from sending messages until a \texttt{view} is delivered to it. In [12], we model this assumption with an abstract application automaton.

The \texttt{gcs\_p} automaton appears in Figure 11. After receiving the first \texttt{start\_change} notification in a given view, end-point \texttt{p} issues a \texttt{block} request to the application and awaits receiving a \texttt{block\_ok} response before sending a synchronization message to other members of \texttt{start\_change.set}. The \texttt{cut} sent in the synchronization message commits to all the messages \texttt{p} received from its application in the current view.

Since the application is required to respond with \texttt{block\_ok} and is then blocked from sending further messages, and since the \texttt{p}'s \texttt{cut} commits to all the messages the application has sent in the current view, the set of messages agreed upon based on the \texttt{cuts} includes all of \texttt{p}'s messages. Therefore, \texttt{p} delivers all these messages before moving on to a new view, and Self Delivery is satisfied. Since end-point \texttt{p} has its own messages on the \texttt{msgs\_p\_p} queue, the modification does not affect the liveness property of \texttt{vs\_rfifo\_ts}. Finally, we note that due to the use of inheritance, the \texttt{gcs\_p} automaton satisfies all the properties we have specified in Section 4.
98    Idit Keidar and Roger Khazan

\textsc{automanon gcs}_p = \textsc{vs_rfifo+ts+sd}_p \quad \text{modifies} \quad \textsc{vs_rfifo+ts}_p

\textbf{Signature Extension:}
\begin{itemize}
  \item \textbf{Input}: \, \texttt{block\_ok}_p() \, \text{new}
  \item \textbf{Output}: \, \texttt{block\_ok}_p() \, \text{new}
  \hspace{1em} \texttt{view}_p(v,T) \, \text{modifies} \, \texttt{vs\_rfifo+ts\_view}_p(v,T)
  \hspace{1em} \texttt{co\_rfifo\_send}_p(set, m) \, \text{modifies}
  \hspace{1em} \texttt{vs\_rfifo+ts\_co\_rfifo\_send}_p(set, \text{SyncMsg} \, m)
\end{itemize}

\textbf{State Extension:}
\begin{itemize}
  \item \texttt{block\_status} \in \{\text{unblocked, requested, blocked}\}, \text{initially unblocked}
\end{itemize}

\textbf{Transition Restriction:}
\begin{itemize}
  \item \textbf{OUTPUT} \, \texttt{block}_p()
    \hspace{1em} \text{pre:} \, \texttt{start\_change} \neq \bot
    \hspace{1em} \text{eff:} \, \texttt{block\_status} \leftarrow \text{requested}
  \item \textbf{INPUT} \, \texttt{block\_ok}_p()
    \hspace{1em} \text{eff:} \, \texttt{block\_status} \leftarrow \text{blocked}
  \item \textbf{OUTPUT} \, \texttt{co\_rfifo\_send}_p(set, tag=\text{sync\_msg}, cid, v, \text{cut})
    \hspace{1em} \text{pre:} \, \texttt{block\_status} = \text{blocked}
  \item \textbf{OUTPUT} \, \texttt{view}_p(v,T)
    \hspace{1em} \text{eff:} \, \texttt{block\_status} \leftarrow \text{unblocked}
\end{itemize}

\textbf{Fig. 11.} \textsc{gcs}_p \text{ end-point automaton.}

\section{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 \textit{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 scalability 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 $\text{VS}_{\text{RFIFO}+\text{TS}}$ simulates $\text{VS}_{\text{RFIFO}+\text{TS}} : \text{SPEC}$ we extend the simulation relation from $\text{WV}_{\text{RFIFO}}$ to $\text{WV}_{\text{RFIFO}} : \text{SPEC}$ 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


