SICS Technical Report T2012:07
ISSN 1100-3154

EFFICIENT SIMULATION OF VIEW SYNCHRONY

August 2, 2012

Frej Drejhammar

Swedish Institute of Computer Science
Box 1263
SE-164 29 Kista, SWEDEN

Seif Haridi

Swedish Institute of Computer Science
Box 1263
SE-164 29 Kista, SWEDEN

ABSTRACT

This report presents an algorithm for efficiently simulating view synchrony, including failure-atomic total-order multicast in a discrete-time event simulator. In this report we show how a view synchrony implementation tailored to a simulated environment removes the need for third party middleware and detailed network simulation, thus reducing the complexity of a test environment. An additional advantage is that simulated view synchrony can generate all timing behaviours allowed by the model instead of just those exhibited by a particular view synchrony implementation.

1 INTRODUCTION

This report describes how view synchrony including failure-atomic total-order multicast in a partitionable network environment can be implemented efficiently in a discrete-time event simulator. View synchrony (Birman and Joseph 1987) is a communications paradigm for building reliable distributed systems (Birman 2006), it is provided by middleware such as JGroups (JGroups 2008), Spread (Amir and Stanton 1998) and FTM (Ventura Networks Inc. 2009). To the best of our knowledge, it is the first description of a view synchrony protocol tailored for a simulated environment.

Testing and debugging a complex application using view synchrony is hard and typically requires a network-level test environment. Using a custom view synchrony implementation, intended for use in a simulated environment, reduces the complexity of the test environment and can be made to exhibit the full timing range allowed by view synchrony instead of the subset occurring in a particular implementation and testing environment.

View synchrony can be implemented concisely by keeping track of view and partition states and using this information to schedule delivery of application messages and middleware events.

1.1 STRUCTURE OF THE REPORT

This report is organized as follows: We start with an introduction to view synchrony in Section 2 In Section 3 we describe the simulator algorithm. Section 4 gives an informal proof that the described algorithm has the properties desired for a group communications system in a partitionable environment, as proposed by Babaoglu et al. (Babaoglu, Davoli, and Montresor 2001). In Section 5 we present related work before concluding the report in Section 6.

2 VIEW SYNCHRONY

View synchrony is a group communications abstraction in which message delivery is uniform[1] within a particular system configuration called a *view* and was first introduced by Birman in (Birman and Joseph 1987).

[1] if a node delivers a message, all non-faulty nodes deliver the message

The semantics of view synchrony as a group communications abstraction in partitionable systems has been formalized by Babaoglu et al. (Babaoglu, Davoli, and Montresor 2001). Friedman and van Renesse (Friedman and van Renesse 1995) extended the model to guarantee that messages are delivered in the same view as they were sent in or not at all. In order to match the functionality provided by the Spread (Amir and Stanton 1998), FTM (Ventura Networks Inc. 2009) and JGroups (JGroups 2008) middlewares we allow unicast messages and additional guarantees of FIFO message delivery and total order broadcasts. The semantics, quoted from Babaoglu et al. (Babaoglu, Davoli, and Montresor 2001) and Friedman and van Renesse (Friedman and van Renesse 1995) but adapted to match our terminology, are:
GM1: View Accuracy If there is a time after which node q remains reachable from some correct node p, then eventually the current view of p will always include q.
GM2: View Completeness If there is a time after which all nodes in some partition Θ remain unreachable from the rest of the group, then eventually the current view of every correct node not in Θ will never include any node in Θ.
GM3: View Coherency (i) If a correct node p installs view v, then either all member nodes of v also installs v, or p eventually installs an immediate successor to v.

(ii) If two nodes p and q initially install the same view v and p later on installs an immediate successor to v, then eventually either q also installs an immediate successor to v, or q crashes.

(iii) When node p installs a view w as the immediate successor to view v, all nodes that survive from view v to w along with p have previously installed v.
GM4: View Order The order in which nodes install views is such that the successor relation is a partial order, i.e. if two views are installed by a node in a given order, the same two views cannot be installed in the opposite order by some other node.
GM5: View Integrity Every view installed by a node includes the node itself.
RM1: Message Agreement Given two views v and w such that w is an immediate successor of v, all nodes belonging to both views deliver the same set of multicast messages in view v.
RM2: Uniqueness Each multicast message, if delivered at all, is delivered in exactly one view.
RM3: Merging Rule Two views merging into a common view must have disjoint compositions.
RM4: Message Integrity Each node delivers a message at most once and only if some node actually multicast it earlier.
RM5: Liveness (i) A correct node always delivers its own multicast messages.

(ii) Let p be a correct node that delivers message m in view v that includes some other node q. If q never delivers m, then p will eventually install a new view w as the immediate successor to v.
Same View Delivery Messages are delivered in the same view as they were sent in. This requirement was introduced by Friedman and van Renesse in (Friedman and van Renesse 1995).

The extended properties, added to match the functionality provided by the Spread, FTM and JGroups middlewares are:
FIFO: FIFO Message Ordering Messages (both unicast and multicast) sent from one node to another node are delivered in the order they were sent (first-in first-out).
TO: Total Order Group-broadcast messages are total order, i.e. they are delivered by all nodes in the same order (total order).

3 VIEW SYNCHRONY SIMULATION

In a real system, FIFO ordering of point-to-point messages is handled implicitly by underlying network layers or by an explicit sequence numbering scheme. View synchronous total order broadcast is implemented by a consensus implementation agreeing on the delivery order of the messages. In a simulated environment, where the complete system state is available, simpler mechanisms can be used to ensure message ordering guarantees.

When the network connectivity is changed due to merging and partitioning this is reflected in the state of the simulator. Partitioning is handled by cloning the current network partition into two new partitions

in which the unreachable nodes belonging to the other partition are treated as if they have crashed. When partitions are merged, a new partition is created and blocking of the views in the parent partitions is triggered. A view is created in the new partition as soon as the views in all parents have been flushed.

Our view synchrony simulator uses a simple network simulator to implement message delivery within views. The main purpose of the network simulator is to track network connectivity and provide reliable message delivery and atomic broadcasts to correct nodes within partitions.

We start the description of our view synchrony simulator by describing the simulator interface in Section 3.1 before describing the semantics of the simple network simulator in Section 3.2. We then describe the notation used in the pseudo-code in Section 3.3 and the simulator state in Section 3.4 before describing the details of intra- and inter-view event scheduling in Section 3.5 and Section 3.6 respectively.

3.1 Simulator Interface

The view synchrony simulator is controlled by the interface described in Table 1 and Table 2. There are three groups of requests, requests from the application running on top of the simulator: start(), send(), mcast() and blockACK(); requests to change the network model: partition() and merge(); and a request stop() which can be used both by the application to leave the network and to change the network model. The user is informed about request progress by the listed events.

Table 1: Simulator Requests

`start(`n`, `p`)` – Start a node n in the partition p. Success will be indicated by a `newView` event, failure by a `retryStart` event.
`stop(`n`)` – Stop node n.
`partition(`p`, `a`, `b`)` – Partition p into two partitions containing node sets a and b respectively.
`merge(`$p0$`, `$p1$`)` – Merge the two partitions $p0$ and $p1$. Success will be indicated by a `mergeOk` event, failure by a `retryMerge` event.
`send(`s`, `d`, `m`)` – Send a point to point message, m, from node s to node d.
`mcast(`s`, `m`)` – Send a multicast from node s to the members of the view.
`blockACK(`n`)` – Have node n acknowledge a `blockRequest`.

Table 2: Simulator Events

`retryStart(`n`)` – Reply to a `start()` operation indicating that it should be retried.
`retryMerge(`$p0, p1$`)` – Reply to a `merge()` operation indicating that it should be retried.
`mergeOk()` – Reply to a successful `merge()` operation.
`deliver(`s`, `d`, `m`)` – Trigger delivery of unicast message m sent from node s on node d.
`deliverMC(`s`, `d`, `m`)` – Trigger delivery of a multicast m sent from node s on node d.
`newView(`n`, `v`)` – Install a view v on node n.
`blockRequest(`n`)` – Request node n to block.

3.2 Network Simulator

The interface to the network simulator is shown in Table 3 and Table 4. The network simulator guarantees that messages will never be delivered to a node which has stopped or restarted since the message was sent. The same applies to events scheduled for a node by $nschedule()$. If a partition p_0 partitions into partitions p_1 and p_2 at time t_p a message sent before t_p by node n, $n \in p_0 \cup p_1$, will be delivered to all nodes in p_1. Delivery to all nodes in p_2 is guaranteed if any node in p_2 had received the message before t_p. If no nodes in p_2 have received the message before t_p the message is either delivered to all members of p_2 or none.

Table 3: Network Simulator Interface Requests

`nstart(`n`, `p`)` – Start a node n in the partition p
`nstop(`n`)` – Stop node n.
`npartition(`p`, `a`, `b`)` – Partition p into two partitions containing node sets a and b respectively. Returns a tuple $(p0, p1)$ with the resulting partitions.
`nmerge(`$p0$`, `$p1$`)` – Merge the two partitions $p0$ and $p1$. Returns the new partition.
`nsendU(`t_d`, `s`, `d`, `m`)` – Send a point to point message, m, from node s for delivery to node d at time t_d.
`nsendM(`t`, `s`, `m`)` – Send a multicast message m from node s for delivery at time t_d on node n_d for each (t_d, n_d) tuple in t.
`nemptypart()` – Return an empty partition.
`partitions()` – Return the set of existing non-empty partitions.
`nreachable(`a`, `b`)` – Return true if node b currently is reachable from node a.
`partof(`n`)` – Return the partition of node n.
`nschedule(`t`, `e`)` – Schedule event e to occur at time t.
`nnodes(`p`)` – Return the set of nodes in partition p.

Table 4: Network Simulator Interface Events

`ndeliverU(`s`, `d`, `m`)` – Trigger delivery of message m sent from node s on node d.
`ndeliverM(`s`, `d`, `m`)` – Trigger delivery of multicast m sent from node s on node d.

3.3 Notation

The pseudo-code describing the simulation algorithm uses a notation inspired by Guerraoui and Rodrigues in (Guerraoui and Rodrigues 2006). We extend their notation to include tuples, written as $(element0, element1, ...)$, and a shorthand notation to access specific fields of a tuple: $v.element$ will access the field named $element$ of the tuple v. Tuples can be pattern matched in which case '_' is used as a wild-card.

The $max()$-function is applicable to both scalars and arrays. Given an array argument it returns the maximum element in the array. The symbol t_{now} is the current simulator time. The constant Δt_ϵ is the smallest non-zero time interval that can be expressed in the simulator.

Algorithm 1 Schedule installation of view v

```
1:  procedure viewInstall(v)
2:      V[v].status := active
3:      V[v].t_l := t_now
4:      V[v].pending := V[v].nodes
5:      for d ∈ nnodes(p) do
6:          t_i := t_now + random()
7:          V[v].t_i[d] := t_i
8:          for s ∈ nnodes(p) do
9:              N[s].fifo[d] := t_i
10:             nschedule(t_i, installView(p, d, v))
11:         end for
12:     end for
13: end procedure
14: upon event installView(p, n, v) ▷ Install view
    v in node n in partition p
15:     N[n].status := unblocked
16:     N[n].v := v
17:     trigger newView(n, v)
18: end event
```

Algorithm 2 Send a message msg from s to d

```
1:  upon request send(s, d, msg)
2:      if !nreachable(s, d) then
3:          return
4:      end if
5:      t_d := max(t_now, N[s].fifo[d]) + random() +
    Δt_ε
6:      N[s].fifo[d] := t_d
7:      V[N[s].v].t_l := max(V[N[s].v].t_l, t_d)
8:      nsendU(t_d, s, d, msg)
9:  end request
10: upon event ndeliverU(s, d, msg)
11:     trigger deliver(s, d, msg)
12: end event
```

3.4 Simulator State

The simulator maintains three state arrays: N, V, and P storing, respectively, the complete state of nodes, views, and partitions. The arrays are indexed by node, view and partition identities respectively.

Each element in the N-array is a node state tuple $(p, v, fifo, status)$ where p is a partition identity indicating a slot in the P-array; v is a view identity and points into the V-array; $fifo$ is an array of time-stamps indexed by node identities and $status$ is a node status from the set $\{blocked, unblocked\}$. The $fifo$ array stores the latest delivery time of a message sent by the node to node d at index d. When a new view is scheduled for installation, the $fifo$-array of each of the view members is initialized to the view installation time at the respective node (Algorithm 1, line 9). The array is updated each time a message is scheduled for delivery.

The V-array consists of view state tuples $(p, t_i, t_l, nodes, lost, state, pending, t_{block})$ where: p is a partition identity; t_i is an array of time-stamps indexed by node identities, it gives the time the view was installed at a given node; t_l is a time-stamp of the latest scheduled event in the view. When a view is installed it is initialized to the latest view installation time of all nodes in the view (Algorithm 1, line 14); $nodes$ is a set of node identities representing the nodes in the view; $lost$ is a set of node identities representing nodes

Algorithm 3 Send a multicast msg from s to all members of the view

1: **upon request mcast**(s, d, msg)
2: $t_{first} := max(t_{now}, V[N[s].v].t_l) + \Delta t_\epsilon$
3: $t_d := \{(d, max(t_{first}, N[s].fifo[d]) + random()) | d \in V[N[s].v].n\}$
4: **for** $(d, t) \in t_d$ **do**
5: $N[s].fifo[d] := t$
6: **end for**
7: $V[N[s].v].t_l := max(\{t | (\_, t) \in t_d\})$
8: $nsendM(t_d, s, msg)$
9: **end request**
10: **upon event ndeliverM**(s, d, msg)
11: **trigger** $deliverMC(s, d, msg)$
12: **end event**

Algorithm 4 Block all nodes in view v

1: **procedure blockView**(v)
2: **if** $V[v].nodes = \emptyset$ **then**
3: $partitionFlushed(V[v].p)$
4: **return**
5: **end if**
6: **if** $V[v].status \in \{pending\_blocked, blocked\}$ **then**
7: **return**
8: **end if**
9: $V[v].status := pending\_blocked$
10: **for** $x \in V[v].nodes$ **do**
11: $t := max(t_{now}, V[v].t_i[x]) + random()$
12: $V[v].t_l := max(V[v].t_l, t)$
13: $nschedule(t,$ **trigger** $blockRequest(x, v))$
14: **end for**
15: **if** $V[v].pending = \emptyset$ **then**
16: $blockComplete(v)$
17: **end if**
18: **end procedure**

that have failed in the view; $state$ is a view state from the set $\{fresh, active, pending\_block, blocked\}$; $pending$ is a set of node identities representing nodes with outstanding blockACK responses and t_{block} is a time stamp of when all events in the view will have been delivered after a block.

The P-array stores partition state tuples (p_v, p_f, p_p) where: p_v is the view identity for the partition, it is nil if a view is yet to be generated; p_f is a partition identity giving the successor to the partition, $p_f = nil$ if there is no successor; p_p is a set of partitions preceding this partition, it is $\emptyset$ for partitions created from scratch.

3.5 Intra-view Event Scheduling

Point-to-point Scheduling To preserve FIFO ordering of point-to-point messages, a point-to-point message from node s to d is scheduled for delivery at time $t_d = t_e + t_r$ where t_e is the first possible delivery time and t_r is a random delay > 0. The first possible delivery time t_e is defined as $t_e = max(N[s].fifo[d], t_{now})$.

Pseudo-code for point-to-point message sending and delivery is shown in Algorithm 2.

Broadcast Scheduling A broadcast message from node s in a view with participating nodes N is handled similarly to scheduling of point-to-point messages but here total order of broadcasts must be ensured. This is accomplished by calculating a first possible delivery time, t_e, for any node in the view, $t_e = max(t_{now}, t_l)$. Then for each node n, a delivery time, t_{d_n}, is calculated as $t_{d_n} = max(t_e, N[s].fifo[n]) + t_r$ with t_r a random delay > 0. Pseudo-code for broadcasts is shown in Algorithm 3.

3.6 Inter-view Event Scheduling

Blocking a View When view v is to be blocked (Algorithm 4), a blockRequest (Algorithm 4, line 13) is scheduled for delivery to all view participants. The blockRequest is scheduled for delivery at node n after a random delay according to $max(t_{now}, t_r, V[v].v_i[n])$ where $v_i[n]$ is the view installation time at node n and t_r is a random delay > 0. The simulator tracks blockACK replies from the participating nodes

Algorithm 5 Response to a blockRequest from node n regarding view v

```
1: upon request blockACK(n, v)
2:     N[n].status := blocked
3:     V[v].pending := V[v].pending \ {n}
4:     if V[v].pending = ∅ then
5:         blockComplete(v)
6:     end if
7: end request
```

Algorithm 6 Try to generate a new view in partition p

```
1: procedure blockComplete(v)
2:     t := max(t_now, V[v].t_l) + Δt_ε
3:     V[v].t_l := max(V[v].t_l, t)
4:     nschedule(t, viewFlushed(v))
5: end procedure
```

Algorithm 7 Start node n in partition p

```
1: upon request start(n, p)
2:     if n ∈ V[P[p].v].lost then
3:         trigger retryStart(n)
4:     else
5:         nstart(n, p)
6:         N[n] := (p, nil, nil, blocked, t_now)
7:         blockView(P[p].v)
8:     end if
9: end request
```

Algorithm 8 Handle a completed flush phase in view v

```
1: procedure viewFlushed(v)
2:     if V[v].p = nil then
3:         return
4:     end if
5:     p := V[v].p
6:     if P[p].next = nil then
7:         v := new_id()
8:         V[v] := (p, nil, t_now, nnodes(p), ∅, fresh, ∅)
9:         P[p].v := v
10:        viewInstall(v)
11:    else
12:        partitionFlushed(P[p].next, p)
13:    end if
14: end procedure
```

Algorithm 9 Stop node n

```
1: upon request stop(n)
2:     p := npartof(n)
3:     nstop(n)
4:     blockView(P[p].v)
5:     if N[n].v ≠ nil AND N[n].status ≠ blocked then
6:         v := N[n].v
7:         N[n] := nil
8:         V[v].nodes := V[v].nodes \ {n}
9:         V[v].lost := V[v].lost ∪ {n}
10:        V[v].pending := V[v].pending \ {n}
11:        if V[v].pending = ∅ AND V[v].status = blocked then
12:            blockComplete(v)
13:        end if
14:    end if
15: end request
```

(Algorithm 5) and when all non-failed nodes have ACKed, the simulator considers the view to be in the flush phase. While waiting for blockACKs the simulator also tracks failed nodes to avoid waiting for an ACK that will never arrive (Algorithm 9, line 10-12). When the flush phase has been entered, the current partition is informed that the view has been flushed at time $max(t_l, t_{now}) + \Delta t_\epsilon$ (Algorithm 6).

Generation of a new view follows a completed flush and is handled by Algorithm 8 which also, if a partition merge has occurred instead informs the successor partition (line 12). Algorithm 8 also handles the case when partitioning has occurred and the flushComplete event should be ignored (line 2-3).

Algorithm 10 Partition p into two partitions consisting of the nodes n_0 and n_1 respectively

```
1:  upon request partition(p, n_0, n_1)
2:      (p_0, p_1) := npartition(p, n_0, n_1)
3:      v := P[p].v
4:      v_0 := project(v, p_0, n_0)
5:      v_1 := project(v, p_1, n_1)
6:      P[p_0].v := v_0
7:      P[p_1].v := v_1
8:      V[v].p := nil
9:      blockView(v_0)
10:     blockView(v_1)
11: end request
```

Starting and Stopping Nodes To handle a starting node the simulator simply triggers blocking of the main view in the current partition (Algorithm 7). A stopped/crashed node is handled by blocking the current view and adding the node to the $lost$ set. If the view is already in the process of being blocked, the simulator updates its state to not expect a blockACK from the stopped node (Algorithm 9, line 10-12). When the view has been flushed a new view will be generated.

In order to reduce the complexity of the simulator and still fulfill RM1 the simulator refuses to start a node, by sending a $retryStart$ event, if the node is in the $lost$ set of the current view for the partition. This behaviour could be hidden by having the simulator remember the merge operation and restart it as soon as a view with the node absent has been generated, but for reasons of simplicity this is not implemented. It is important to note that any correct view synchrony protocol would delay delivery of a view to the newly started node and that automatic retry is, from the point of view of the application, indistinguishable from a delayed view installation.

Network Partitions When a network partitions, the main view is cloned into two new views, which become the main views for the new partitions. The simulator state associated with each of the cloned views is updated to treat the nodes in the other partition as if they have stopped (added to the $lost$-set). A view block is then triggered in both views which will, when the respective view has been flushed, trigger the generation of new views. The full procedure is shown in Algorithm 10, with the view update shown in Algorithm 12.

Merging of Network Partitions When two partitions are merged (Algorithm 11), a block is triggered in their respective main views, if they are not already in the flush phase. Additionally the new resulting partition is annotated with information about its preceding partitions (p_p) and the forwarding pointers (p_f) in the old partitions are set to point to the new partition as implemented by Algorithm 8 and Algorithm 13. The forwarding pointers and the information about the parents are used to ensure that a new view is only generated when both parents have flushed their views.

A merge will be refused if the current views of the two partitions have non-disjoint compositions to avoid violating RM3. The composition will be disjoint if member nodes in one partition are not members of the $lost$-set in the current view for the other partition and vice versa. Just as for node starts a refused join could be automatically retried as soon as views taking the lost nodes into account have been generated.

View Generation A new view can be generated and scheduled for installation in the partition if the views in the parent partitions are flushed and the current partition has not been superseded, i.e. $p_f \neq nil$. The new view includes all nodes of the current network partition.

If the current partition is informed that a parent partition, a partition in the set p_p, has completed the flush phase it is removed from p_p. If $p_p \neq \emptyset$ nothing more is done. If $p_p = \emptyset$ and p_f is set, the current view has been superseded and partition p_f is informed that the current partition has been flushed (Algorithm 13 and Algorithm 8). If the current partition has not been superseded a new view is generated in the partition.

Algorithm 11 Merge partitions p_0 and p_1

1: **upon request merge**(p_0, p_1)
2: **if** $V[P[p_0].v].lost \cap nnodes(p_1) \neq \emptyset$ **OR** $V[P[p_1].v].lost \cap nnodes(p_0) \neq \emptyset$ **then**
3: **trigger** $retryMerge(p_0, p_1)$
4: **else**
5: **trigger** $mergeOk()$
6: $ns := nnodes(p_0) \cup nnodes(p_1)$
7: $p := nmerge(p_0, p_1)$
8: $parents := \emptyset$
9: **if** $V[P[p_0].v].status \neq blocked$ **then**
10: $parents := parents \cup \{p_0\}$
11: **end if**
12: **if** $V[P[p_1].v].status \neq blocked$ **then**
13: $parents := parents \cup \{p_1\}$
14: **end if**
15: $P[p] := (nil, nil, parents)$
16: $P[p_0].next := p$
17: $P[p_1].next := p$
18: **for** $\forall n \in ns$ **do**
19: $N[n].p := p$
20: **end for**
21: **if** $V[P[p_0].v].status \neq blocked$ **then**
22: $blockView(P[p_0].v)$
23: **end if**
24: **if** $V[P[p_1].v].status \neq blocked$ **then**
25: $blockView(P[p_1].v)$
26: **end if**
27: **end if**
28: **end request**

Algorithm 12 Create a new view and update it to reflect that only the nodes in ns are present in the partition

1: **procedure project**(v, p, ns)
2: **for** $\forall n \in ns$ **do**
3: $nodes[n].p := p$
4: **end for**
5: $(\_, t_i, t_l, nodes, lost, state, pending, t_{block}) := V[v]$
6: $v_n := new\_id()$
7: $V[v_n] := (p, t_i, t_l, \{x | x \in N, x \in ns\}, lost \cup (N \setminus ns), state, \{x | x \in pending, x \in ns\}, t_{block})$
8: **if** $t_{block} \neq -1$ **then**
9: $V[v_n].t_l := max(V[v_n].t_l, t_{block})$
10: $nschedule(t_{block}, viewFlushed(v_n))$
11: **end if**
12: **return** v_n
13: **end procedure**

Algorithm 13 Report that partition p_0 has been flushed to partition p

```
1:  procedure partitionFlushed(p, p_0)
2:      P[p].parents := P[p].parents \ {p_0}
3:      if nnodes(p) ≠ ∅ AND P[p].parents = ∅ then
4:          for s ∈ nnodes(p) do
5:              N[s].p := p
6:          end for
7:          v := new_id()
8:          V[v] := (p, t_now, t_now, nnodes(p), fresh, ∅)
9:          viewInstall(v)
10:     end if
11: end procedure
```

When scheduling the view installation (Algorithm 1), the $fifo$ arrays and t_l timestamp are initialized as described in Section 3.4.

4 INFORMAL CORRECTNESS PROOF

In Section 2 we give a list of properties which are considered necessary for a practically usable communications service. In this section we give an informal proof that the simulation algorithm described in Section 3 has the same properties. Each subsection starts by the property definition from Section 2 in italics. When reasoning about state variables changing over time we use the notation $v(n)$ which is defined as the value of v at time step n.

GM1: View Accuracy View generation is triggered by the end of the flush phase, the flush phase is in turn triggered by completion of the block phase and blocking is initiated by changes in connectivity. As view creation includes all members of the partition in the generated view, eventually all nodes reachable from p will be included in the view.

GM2: View Completeness Using the same reasoning as for GM1 we can conclude that all nodes not in Θ will install a view in which the members of Θ are no longer present.

GM3: View Coherency *(i)* When a view, v, is created, an installation event is created for each view member. If no failures occur the view will be installed on all members. If failures occur, the simulator will block v and then generate and install a new view which will be the immediate successor to v . The described behaviour of the simulator provides the desired property, as if a view is installed on one node it will be installed on all correct member nodes and the existence of faulty nodes will trigger the installation of a successor view.

(ii) View installation events for a successor view are scheduled by the simulator, for all member nodes, when a view has been flushed. Two correct nodes having installed the same view will therefore also install a successor view, the only way for one of them to not install a successor view is if it stops.

(iii) For a new view to be scheduled for installation, the previous view must be flushed. To enter the flush phase the view must in turn have been blocked. Blocking requires all view members to acknowledge the block. By construction, the block request for a view cannot arrive at a node before the view is installed. The simulator also never removes correct nodes from a view when its successor is created. Therefore we are assured that when w is installed on p as the direct successor to v all surviving members of v in p's partition will also have installed v.

GM4: View Order The simulator enforces the installation order of views by not generating and installing a successor view until all previous views in the current partition have been flushed. As the installation of a view on all member nodes is a prerequisite for the completion of the flushing phase this implies that at most one view is pending installation at any time in a given partition. Given that only at most a single view is pending installation at any given time, two nodes cannot observe different view installation orders.

GM5: View Integrity By construction, view installation events for a view are scheduled for all members of the view.

RM1: Message Agreement The simulator schedules multicast delivery events for all correct nodes in the view. The simulator will never drop a multicast message scheduled for delivery to a node unless the node leaves the partition or crashes. A successor view is not installed until all its predecessors have completed flushing. Nodes staying in the same partition will not be removed from the view when the immediate successor is created. This implies that all nodes belonging to the same view will have delivered the same set of messages in the previous view.

RM2: Uniqueness Multicast messages are scheduled for delivery after the installation, at the respective destination node, of the view in which they were sent. Installation of a new view requires the flushing of the previous view. The simulator schedules the completion of the flush phase after all messages sent in the view have been delivered, using the t_l field, and therefore multicast messages can only be delivered in exactly one view.

RM3: Merging Rule Views are only merged in response to merging of partitions. As the simulator refuses to complete a merge when the resulting view would have a non-disjoint composition this property trivially holds.

RM4: Message Integrity This property is trivially fulfilled by the simulator as messages are only scheduled for delivery in response to message sending and only one delivery event is created for each destination node.

RM5: Liveness *(i)* When a multicast messages is sent, a delivery event is scheduled for each member of the view including the sender itself. Unless the sender stops it will not leave its partition and therefore it will always receive its own multicast messages.

(ii) Given that m is delivered to p in view v, which includes node q, we know that a delivery event for m at node q was scheduled as well. As the only way in which m will not be delivered to q is if q leaves the partition, due to a stop or partition request. We are therefore assured that a new view will be generated and eventually installed as the successor to v on all correct nodes, including p.

FIFO: FIFO Message Ordering The delivery time, t_d, of a message sent from node s at time t_s to d is for point-to-point messages defined as $t_d = max(N[s].fifo[d], t_{now}) + t_r$, where $t_r > 0$ (Algorithm 2). For broadcasts it is defined as $t_d = max(t_{now}, t_l, N[s].fifo[d]) + t_r$ (Algorithm 3). Both broadcasts and point-to-point messages update $N[s].\ fifo[d]$ to the new t_d. For point-to-point messages t_l is updated according to $t_l(n+1) := max(t_l(n), t_d)$. Broadcasts also update t_l to the largest t_d , called $\hat{t}_d$, encountered when scheduling delivery to all nodes in the view according to $t_l(n+1) := max(t_l(n), \hat{t}_d)$. As $t_r > 0$ we can conclude that $N[s].fifo[d](n+1) > N[s].fifo[d](n)$ and $t_l(n+1) \geq N[s].fifo[d](n+1)$.

For both point-to-point messages and broadcasts $N[s].fifo[d]$ is strictly increasing for the pair of communicating nodes. As $N[s].fifo[d]$ is equal to the delivery time of the latest sent message we can conclude that the delivery order of point-to-point messages is FIFO. Given that the broadcast delivery time to node d is $max(t_{now}, t_l, N[s].fifo[d]) + t_r$ and $t_l(n) \geq N[s].fifo[d](n)$ we can also conclude that broadcasts sent from one node are delivered to their destination in FIFO order.

TO: Total Order The first possible delivery time, t_e, of a broadcast message is $max(t_{now}, t_l)$. When a broadcast has been scheduled for delivery on all nodes, as $t_{d_n} = max(t_e,\ N[s].fifo[n]) + t_{r_n}$ for each node n in the view, t_l is updated according to $t_l(n+1) := max(t_l(n), \hat{t}_d)$ where $\hat{t}_d = max(\{t_{d_n} | n \in N\})$ i.e. the timestamp of the latest delivery of the message. Given that $t_r > 0$ we know that the $(n+1)$-th broadcast message will be delivered to all members of the view within the time interval $(t_l(n), t_l(n+1)]$. As $t_r > 0$ we can conclude that, for broadcasts, $t_l(n+1) > t_l(n)$ meaning that the delivery intervals are disjoint. As the delivery intervals are disjoint, it is not possible for two different broadcasts to be delivered in a different order at different nodes.

Same View: Sending and Delivery in the same View The flush phase starts when all nodes have acknowledged the block request. Nodes do not send messages when in the flush phase. By construction the flush phase does not complete until all messages sent in the view have been delivered. As installation

of a new view is required for sending further messages, and the view will not be installed until the flush phase completes, we can be sure that all messages are delivered in the view in which they were sent.

5 RELATED WORK

The authors are not aware of any algorithms for simulating view synchrony where the availability of global state is used to simplify the implementation and increase the efficiency of the simulator.

6 CONCLUSION

The main advantage of simulating view synchrony instead of implementing one of the several known algorithms on top of a network simulator is that it reduces the complexity of the testing environment. Additionally, simulated view synchrony allows an application/protocol to be exposed to all timing behaviours allowable under view synchrony and not just those exhibited by a particular middleware implementation.

References

Amir, Y., and J. Stanton. 1998. "The Spread Wide Area Group Communication System". Technical Report CNDS-98-4, Johns Hopkins University.

Babaoglu, O., R. Davoli, and A. Montresor. 2001, April. "Group Communication in Partitionable Systems: Specification and Algorithms". *IEEE Transactions on Software Engineering* 27 (4): 308–336.

Birman, K. 2006. *Reliable Distributed Systems Technologies, Web Services, and Applications*. Springer.

Birman, K. P., and T. A. Joseph. 1987. "Reliable communication in the presence of failures". *ACM Trans. Comput. Syst.* 5 (1): 47–76.

Friedman, R., and R. van Renesse. 1995. "Strong and Weak Virtual Synchrony in Horus". Technical Report TR95-1537, Cornell University.

Guerraoui, R., and L. Rodrigues. 2006. *Introduction to Reliable Distributed Programming*. Heidelberg, Germany: Springer.

JGroups 2002-2008. "http://www.jgroups.org".

Ventura Networks Inc. 2009. "http://www.venturanetworksinc.com/".