A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs*

Idit Keidar
Lab for Computer Science
Massachusetts Institute of Technology

Jeremy Sussman
Department of Computer Science and Engineering
University of California, San Diego

Keith Marzullo
Department of Computer Science and Engineering
University of California, San Diego

Danny Dolev
Computer Science Institute
The Hebrew University of Jerusalem, Israel

Abstract

We describe a novel scalable group membership service designed explicitly for wide area networks. Our membership service is scalable in the number of groups supported, in the number of members in each group, and in the topology each group spans. Our service also supplies the hooks needed to provide clients with full virtual synchrony semantics. Our service attains, on average, a low message overhead by agreeing on membership within a single message round. Furthermore, our service avoids notifying the application of obsolete membership views when the network is unstable, yet it converges when the network has stabilized.

1 Introduction

Group communication [1, 24] is a means of arranging processes into multicast groups. Group communication has proven to be a useful abstraction in the development of highly available distributed and communication-oriented applications. The most important aspects of this abstraction are the dynamic maintenance of group membership and the interleaving of notifications of group membership changes within the delivery order of multicast messages. There are many diverse semantics for such interleaving.

The task of a *group membership service* is to maintain a list of the currently active and connected processes, which is called the *membership*. When the membership changes, it is delivered to the application at an appropriate point in the delivery sequence. The output of the membership service is called a *view*, consisting of the list of the current members in the group and a unique identifier. Reliable multicast services that deliver messages to the current view members complement the membership service.

Group communication systems are especially useful for constructing fault-tolerant applications that consistently maintain replicated state of some sort, e.g., [13] (see [1, 24] for discussion of the utility of group communication systems). Such applications greatly benefit from *virtually synchronous* communication semantics [19, 14, 24] that synchronize membership notifications with regular messages and thus simulate a "benign" world in which message delivery is reliable within the set of connected processes. A vital part of any virtually synchronous communication service is the membership service, since agreement on uniquely identified views is necessary for synchronizing communication in such views.

The design of a scalable membership service for a wide area network (WAN) is a challenging task. Message latency can be high and unpredictable, making the exchange of information costly. Failure detection in a WAN is usually less accurate than failure detection in a local area network (LAN). This may lead to frequent (and often costly) view changes. Furthermore, there is no efficient support for the flooding of messages in a WAN. A group membership service supporting multiple groups in a WAN must take care not to flood the network.

In this paper, we present a group membership algorithm that is designed for supporting hundreds of clients in WANs. In contrast to previously suggested WAN-oriented group membership services, ours does not evolve from LAN-oriented membership algorithms: it is designed explicitly for the WAN environment.

*Supported by DOD-ARPA contract number F30602-96-1-0313 and by the Israel ministry of Science grant number 1230-1-98.

2 Features

We now discuss the key features of our membership algorithm.

2.1 Avoiding delivery of obsolete views

Further changes in the network connectivity can occur while the membership service is reaching agreement on prior changes. Such concurrent changes are more likely in a WAN due to the inaccuracy of failure detection. Existing group membership algorithms (for example, [10, 11, 2, 22, 14]) can have the current invocation of the membership algorithm proceed to termination without reflecting the new changes, and then invoke the membership algorithm again to reflect the new changes.

Membership changes cause extra overhead for applications that rely on virtual synchrony. For such applications, a view change causes the application to send special messages to re-synchronize their shared state (for examples, please see [24]). Such additional communication is especially costly in WANs.

To avoid such excessive communication and execution penalties, our algorithm does not deliver obsolete views to an application. The membership service waits for agreement among all of the view members about what the view should be. It neither delivers a view without such agreement, nor does it deliver an obsolete view when it has new information that the membership has changed.

By avoiding the delivery of obsolete views, our membership algorithm may be non-terminating if the network does not stabilize, i.e., if the network situation constantly changes. There may be applications that would prefer to try to track the changes in membership at times when the membership is constantly changing. However, such applications would not attempt to re-synchronize their states following view changes. Applications like these typically do not make use of virtual synchrony semantics. A membership algorithm providing virtual synchrony is overkill for such applications, which could be usually satisfied with a network event notifications service (cf. Section 3.1).

2.2 A single round algorithm

Since message latency in WANs may be large, we have designed our membership algorithm to minimize the number of messages exchanged among the servers. In most cases, once a change in network connectivity is detected, each server simply multicasts a message to the other servers. Thus, if the maximum message latency in the network is δ, then the membership algorithm usually terminates within δ time after all of the servers detect the change in connectivity.

If temporary lack of symmetry or transitivity in the network causes surviving members to differ too much in their detections of failures and reconnections, though, then our algorithm may be required to run a re-synchronization round among the servers. In this case, the algorithm terminates within at most 3δ time once network stabilization occurs and all of the servers correctly detect the network connectivity.

After agreement among the servers is reached, each server reports the view to its local clients. Clients do not directly communicate with other servers. Since each client can be served by a server that is proximate to it (preferably in the same LAN), the amount of communication that spans multiple LANs is limited, and depends solely on the number of servers.

2.3 WAN based architectural design

Our membership algorithm is part of a novel architecture for group membership services [5] designed for CSCW and groupware applications in WANs. Two salient features of this architecture are the use of a *network event notification service* as a building block, and the use of a client-server approach.

Group membership algorithms designed for local-area networks often incorporate failure detection into their functionality. Failure detection in a WAN is a difficult problem, and does not scale easily. Therefore, our membership service does not explicitly attempt to detect failures using time-outs. Rather, it uses a *network event notification service* as a building block (cf. Section 3.1). Our algorithm was designed for use with CONGRESS [4] which is a distributed network event notification service geared to WANs.

Our architecture employs a client-server approach: dedicated membership servers maintain process-level group membership (i.e., which clients are members of each group). Server-level group membership is not maintained. The membership servers are only concerned with membership maintenance, and not with message transmission within groups. This architecture allows us to be scalable in the number of groups and in the number of clients.

The membership service interface provides the hooks for clients to efficiently implement virtually synchronous communication semantics, but it does not impose such semantics. Thus, the membership service does not delay delivery of views to clients until such semantics are achieved. It also allows an application to choose, on a per message stream basis, whether to use the network notification service directly or the full membership semantics. Details on this architecture and its utility can be found in [5]. The membership server interface is presented in Section 4.

3 The environment model

Our membership algorithm is implemented in an asynchronous message-passing environment: processes communicate solely by exchanging messages. There is no bound on message delivery time. Processes fail by crashing, and may later recover. Communication links may fail and recover.

3.1 Network event notification service

Our membership service extends a distributed network event notification service such as CONGRESS. The notification service accumulates and disseminates failure detection information, along with information about processes requesting to join or leave groups.

Each membership server has a local notification service component that reports the client status to the membership servers. The notification service reports network events to the membership server via *notification events* (`NEs`), with the following interface:

`NE(`G`, joining, leaving)` is a notification that the processes in the set `joining` are joining group G, and those in the set `leaving` are leaving the group.

A `NE` can report of changes in more than one group by providing a list of triples of the form `<G, joining, leaving>`.

Note that the notification service does not distinguish between processes leaving the group due to failures and processes leaving the group voluntarily. Both are reported via the same interface.

Our membership servers keep track of the membership according to the notification service in a variable called the `NSView`. The `NSView` of a group G is computed by aggregating all of the `NEs` that correspond to G. Note that the `NSView` is not a membership view, since it has no unique identifier that can be agreed upon. The `NSView` is simply the list of group members that are currently not suspected.

As a failure detector in an asynchronous environment, the notification service is bound to be unreliable in some runs [8]: it may be inaccurate in that it may suspect correct processes. However, we assume that the notification service is always *complete*, i.e., it eventually suspects all permanently faulty or disconnected processes.

3.2 Communication guarantees

The reliable FIFO communication layer guarantees that messages from a single source are not received out of order. For example, if process p first sends message m_1 to process q and later sends m_2 to q, and if q delivers both m_1 and m_2, then q delivers m_1 before m_2. In addition, the underlying reliable FIFO communication layer guarantees liveness in conjunction with the notification service. In particular, if server $S1$ sends a message m to server $S2$ at time $t1$, then there is a time $t2 > t1$ by which either $S2$ has received m, or the `NSView` of $S1$ does not contain any client of $S2$.

Such a reliable communication service can be easily implemented by retransmitting lost messages to live processes as long as they are not suspected by the notification service. Group membership algorithms are often based on such services, e.g., in Transis [11], Ensemble [16] and the algorithm described in [6].

4 Membership algorithm guarantees

We now describe the interface the membership algorithm provides to its clients, and the guarantees it makes. The clients use the notification service interface directly to issue join and leave requests. The primary function of our membership algorithm is to provide clients with views that contain a membership and a unique identifier. Each membership server communicates with its clients using reliable FIFO links.

The server sends two types of events to its clients:

`startChange`$(G, startChangeNum, suggestedMemb)$ indicates to the client that the server is now engaging in a membership change for group G.

`view`(G, V) notifies the client that the new view of group G is V. The view V is a triple: `<id, members, startChangeNums>`, where the `id` is an integer and `members` is a set of processes.

The `startChange` event and the third value of a view `V` are used in the implementation of virtual synchrony (see [18] for details).

4.1 Membership guarantees

We say that two processes deliver the same view in a group G if they deliver identical triples. Views are *partially ordered* according to their `id`. The membership algorithm guarantees that the `id`s of views delivered to each client are monotonically increasing:

> **View Identifier Local Monotonicity** If a process delivers a view $V1$ and later delivers a view $V2$, then $V2.\mathtt{id} > V1.\mathtt{id}$.

One of the tasks of a membership service is to reach agreement on views that correctly reflect the network connectivity. Unfortunately, such a desirable membership service is impossible to implement in asynchronous environments [24]. An unstable communication layer can force every deterministic membership algorithm to either block or to constantly deliver changing views. Therefore, we formulate the **Agreement**

on Views property to guarantee only that agreement be reached in runs in which the network stabilizes and the notification service does not suspect correct and connected processes:

> **Agreement on Views** Let G be a group, CS a set of clients, and SS the set of servers serving clients in CS. Assume that there is a time t_0 such that from time t_0 onwards, the `NSView` of G at all of the servers in SS contains exactly the clients in CS. Then eventually, all of the clients in CS receive the *same* `view` V from their servers in group G such that V.`members` $= CS$, and do not receive new `view` or `startChange` messages in group G henceforward.

This property classifies runs in which all of the connected members of G *agree* on the same view forever. Since our algorithm runs in asynchronous systems, it is impossible to guarantee that such agreement be reached in every run. However, such agreement is reached if the following two conditions hold: 1. the set of members of G in a certain connected network component[1] eventually stabilizes; and 2. the notification service behaves like an *eventually perfect* failure detector (cf. [8, 24, 6]), i.e., it eventually stops making mistakes. A similar guarantee is formally defined in terms of network stability and failure detector properties in [6, 24]. For the sake of simplicity, we have summarized both conditions into one requirement.

Note that although the **Agreement on Views** property is guaranteed to hold only in certain runs, the conditions on these runs are *external* to the implementation and therefore cannot be met trivially.

Note also that we require stability to last forever. In practice, it only has to hold long enough for the membership algorithm to execute and for the failure detector module to stabilize, as explained in [12]. This time period depends on external conditions: message latency, process scheduling and processing time. Our membership algorithm typically terminates one message round after such stabilization occurs, and in the worst case 3δ time after such stabilization occurs, where δ is the maximum message latency in the run. However, we do not formally analyze the actual length of time required for the algorithm to terminate. This can be done, as in [13, 9], by explicitly linking the guarantees of the membership algorithm to predetermined bounds on process scheduling times and network delays.

The `startChange` messages and `startChangeNums` are used by the clients for implementing virtual synchrony (see [18]). In order to be useful, `startChangeNums` have to be monotonically increasing and the `startChangeNums` included in views have to correspond to those conveyed in the preceding `startChange` events.

5 The membership algorithm

In this section we discuss the implementation of the group membership algorithm. For simplicity we discuss the membership algorithm for a single group and omit the group name.

The membership algorithm is invoked whenever it receives a `NE`. The typical message flow of the membership algorithm is as follows: On receiving a `NE` from the notification service, the server notifies its clients that the membership is undergoing a change via `startChange` messages. The server also multicasts a `proposal` message to all of the other servers so the servers can agree on the unique identifier of the view to be installed in a manner consistent with the **View Identifier Local Monotonicity** property. Once the server has received a `proposal` from each of the servers, it computes the new view and sends a `view` message to its clients.

A one round algorithm such as this may reach agreement in a failure-free case, but cannot successfully reach agreement under all conditions. Consider Example 5.1 below:

Example 5.1 *Some client c tries to join the group, but fails soon after requesting the join. In such a case, the notification might send a* `NE` *that reflects c joining the group to the server s that is responsible for c. The notification service may later send to s another* `NE` *that reflects c leaving the group. Because c failed so soon after attempting to join the group, the notification service at another server s' might not send a* `NE` *at all. In such a case, s has begun the algorithm and sent a* `startChange` *message to its clients, but s' is not running the algorithm. Thus, s will block waiting for a* `proposal` *from s' that will never be sent, and the algorithm will never terminate.*

Such cases, albeit rare, need to be addressed. Our algorithm is therefore composed of a *fast agreement algorithm* that terminates in one round in the best case, a mechanism for detecting if the fast agreement algorithm is blocked, and a *slow agreement algorithm*

[1] A connected network component is a set of processes among which all of the links are operational and all of the links to processes outside the component are not operational. The existence of such a component implies that communication is transitive and symmetric.

that terminates in all cases. These pieces of the algorithm are presented in pseudo-code in the rest of this section.

5.1 The Fast Agreement Algorithm

The algorithm uses the following variables: The variable `running` is used to track which algorithm is currently being run: its value is initially `none` since no algorithm is running. Its value can also be `FA` for the fast agreement algorithm or `SA` for slow agreement. The set `NSView`, initially empty, contains the aggregation of the `NEs` received from the notification service. The buffer `props` is used to store the most recent `proposal` message received from every server. The variable `curView` contains the most recent view sent to the clients, and is initially a special *initial view*. The integer `startChangeNum` ensures that every `startChange` message sent to a client have a monotonically increasing number, and the integer `propNum` is a logical timestamp used to ensure that every `proposal` message sent by a server has a unique monotonically increasing identifier. The function `usedProps` is used to detect if the fast agreement algorithm is blocked, as described in Section 5.2 below. These last two variables are not used by the fast agreement algorithm.

We assume the existence of two external functions: `serversOf` that maps a set of clients to the set of servers serving those clients, and `local` that maps a set of clients to the subset of those clients being served by this server. These functions can be implemented by using a *naming convention* that associates clients with their local servers. Alternatively, a client can be assigned to a server the first time the client issues a join request, and this information can be disseminated to maintain a registry of the clients.

The algorithm as we have described it herein does not allow a client to be served by more than one server. This implies that when a server crashes, all its local clients are also considered to have crashed. In order to avoid this undesirable situation, we have devised a mechanism that allows fail-over of clients to alternative servers in case of crash. The discussion of this mechanism is not in the scope of this paper.

The membership algorithm is event-driven, and responds to events as they occur. We assume that event handlers are *atomic*, i.e., they cannot be preempted once they are invoked. The algorithm responds to two types of events: the reception of `NEs` from the notification service, and the reception of `proposal` messages that were sent by other servers. The event handlers are presented in Figure 1. Code shown in gray is not part of the fast agreement algorithm.

Upon receiving a `NE` from the notification service, every server sends a `startChange` message to its clients and sends a `proposal` message to all of the servers in the group. The `proposal` message has three fields used by the fast agreement algorithm: `sender` is the server that sent the `proposal` message; `members` indicates the `NSView` that this `proposal` message is proposing for the new view; and `startChangeNum` is used to compute the identifier of the new view, as explained below.

To ensure that the **View Identifier Local Monotonicity** property is not violated, the identifier of the new view must be greater than the identifier of the last view for every client in the new view. The servers use `startChangeNums` to calculate such an identifier: The `startChangeNum` at a server is always greater than or equal to the identifier of the last `view` sent to the clients, and it is included in the `proposal` message. When a server has collected `proposal` messages from all of the servers, it uses the `startChangeNum` values to calculate a new view number greater than all of the previous view numbers.

Reaching agreement on a view is determined via `proposal` messages sent by all of the servers of clients in the `NSView`. The `props` buffer collects these `proposal` messages. Whenever a `proposal` is received, it is placed in the `props` buffer regardless of the membership it proposes. Because of FIFO communication, this `proposal` message is guaranteed to have been sent after the `proposal` message it replaces. By using the most recent `proposal` message sent by the servers, the algorithm avoids delivering obsolete views.

Once a server has received `proposal` messages proposing the same `NSView` from each server that has clients in the `NSView`, the server sends a `view` to its clients. After a `view` is sent to the local clients in C, for each server s of a client in C, `props[s]` is set to `null` in order to avoid using the same `proposal` in future invocations of the membership algorithm. When `props[s]` is set to `null` and `view` V is sent, we say that the `proposal` message that was in `props[s]` was *used* for V.

5.2 The detection mechanism

To satisfy **Agreement on Views**, our membership algorithm must terminate when the network and the `NSView` eventually stabilize. Unfortunately, as illustrated in Example 5.1 above, the fast agreement algorithm may not terminate successfully in some cases, even if the network and `NSView` eventually stabilize. We refer to the failure to terminate as *blocking*.

Blocking stems from transient conditions in the network, such as a lack of symmetry or transitivity in the communication system. Such conditions may cause

```
On receive NE n:
   NSView = NSView ∪ n.joining \ n.leaving    // Update NSView

   if ( local(NSView) ≠ {} ) then // We are only interested in groups that contain local clients
      startChangeNum = max( curView.id, startChangeNum + 1 )
      send startChange ⟨startChangeNum, NSView⟩ to local(NSView)
      running = FA
      propNum = max( propNum, props[serversOf(NSView)].propNum ) + 1
      proposal p = ⟨me, NSView, startChangeNum,  FA, usedProps[serversOf(NSView)], propNum ⟩
      send p to serversOf(NSView) \ {me}
      deliver p immediately to myself // Invoke proposal handler
   endif

On receive proposal inProp:
   props[inProp.sender] = inProp          // Overwrite to use latest proposal (we assume fifo links)

   if ( inProp.members = NSView ) then    // Proposal is only acted upon if it matches the NSView
      if ( TestIfSAProposalNeeded(inProp) ) then SendSAProposal(inProp) endif
      if ( TestIfAgreeementReached() ) then
         curView = ⟨max( props[serversOf(NSView)].startChangeNum ) + 1, NSView,
                    props[serversOf(NSView)].startChangeNum⟩
         forall s ∈ serversOf(NSView)
            usedProps[s] = props[s].propNum
            props[s] = null
         end forall
         running = none
         send curView to local(NSView)
      endif
   endif

// In the fast agreement algorithm:
TestIfAgreeementReached() ≜ ∀s ∈ serversOf(NSView) : props[s].members = NSView
```

Code shown in gray is not part of the fast agreement algorithm.

Figure 1: Event handlers for the membership algorithm.

the servers to receive different sets of network events. Once the network stabilizes, all of the servers will send their last proposal message for the fast agreement algorithm. These proposal messages will all have the same membership. However, some servers may have sent previous proposal messages with the same membership. Since the fast agreement algorithm is a one round algorithm, there is no means of determining which proposal messages are the last ones sent. Thus, one server may use an obsolete proposal message sent by another server along with its own latest proposal message, or vice versa.

Note that we are only interested in detecting non-termination of the fast agreement algorithm in case the network and the NSView eventually do stabilize. If an invocation of the membership algorithm is followed by another NE, then the membership algorithm is re-started and we are no longer concerned with the termination of the former invocation.

The detection mechanism is implemented in the function TestIfSAProposalNeeded, which is invoked whenever a proposal message inProp is received by some server s, as shown in gray in the event handler of Figure 1. The detection mechanism is presented in Figure 2.

```
TestIfSAProposalNeeded(proposal inProp)
   return ( running = none ∨
     inProp.usedProps[me] = propNum )
```

Figure 2: Detecting if the fast agreement algorithm is blocked.

In [17] we prove that whenever the fast agreement algorithm blocks, it is detected by the detection mechanism at some server. We also prove that the detection mechanism only detects when the fast agreement does indeed block.

5.3 The slow agreement algorithm

We have seen that the fast agreement algorithm can block. This blocking is inevitable since a one round algorithm in which all of the servers send messages simultaneously cannot synchronize different invocations of the algorithm. Such synchronization would require all of the servers to use an agreed *round (invocation) number*. However, the algorithm cannot assume that such an agreed round number exists a priori. Another level of knowledge is required to agree on a common round number.

The slow agreement algorithm is begun by a server when it detects that the fast agreement algorithm will not terminate. As with the fast agreement algorithm, in the slow agreement algorithm servers send `proposal` messages to each other and collect these `proposal` messages to agree upon a new view. However, in contrast to the fast agreement algorithm, the invocations of the slow agreement algorithm are synchronized: the set of `proposal` messages used for a view must all carry the same `propNum`. Since each server sends no more than one `SA proposal` with the same `propNum`, if two servers use a `proposal` message p for a view V, then the same set of `proposal` messages are used for V by both servers.

A server that detects blocking of the fast agreement algorithm initiates the slow agreement algorithm by multicasting a `proposal` message to all of the other servers with the `type` field set to `SA`. The `propNum` of this `proposal` is chosen to be *greater than* the maximal value of `propNum` of any `proposal` message (of any type) this server has previously sent, and at least as large as any received `proposal`. This is the *round number* associated with this invocation of the slow agreement algorithm.

Every server that receives a `proposal` of type `SA` while it is not running the slow agreement algorithm joins the slow agreement algorithm by also sending a `proposal` message of type `SA`. A server that joins the slow agreement algorithm sends a `proposal` with the value of `propNum` *equal to* the maximal value of `propNum` in any `proposal` message it previously sent or received. Ideally, this value will be equal to the `propNum` in the initiator's `proposal`[2], and all of the servers will send `proposal` messages with identical `propNum` values.

[2] If the initiator receives the last fast agreement algorithm `proposal` sent by each of the other servers before invoking the slow agreement algorithm, then the `propNum` of its `SA proposal` is greater than the local values of `propNum` at all of the other servers.

However, if the joining server sends a `SA proposal` with a greater `propNum` than the initiator, the rest of the servers (including the initiator) will also have to send `proposal` messages with the higher `propNum` so that the algorithm will be able to terminate. To this end, if a server that has already started (or joined) a round of the slow agreement algorithm receives a `proposal` with a higher `propNum` value than its local one, it joins the higher round by setting its local `propNum` to the higher value and sending a new `SA proposal` with the value.

Note that we do not assume that there is a single initiator. The difference between starting a round of the slow agreement algorithm as an initiator and joining a round is that servers joining a round of the slow agreement algorithm do not increase the `propNum` to be larger than the highest value they received. Thus, when all of the servers are running the slow agreement algorithm, the maximum `propNum` of all of the servers will not increase. This way, all of the servers eventually send `proposal` messages with the same `propNum`. Once `proposal` messages with identical identifiers are collected from all of the servers, a view is sent to the clients and the slow agreement algorithm terminates.

In Figure 3, we complete the pseudo-code shown in Figure 1 by adding the functions that implement the slow agreement algorithm. Recall that if the fast agreement algorithm is detected as blocking, then the slow agreement algorithm is initiated by call of the function `SendSAProposal` at the initiator (cf. Figure 1).

The function `SendSAProposal` is also used by the slow agreement algorithm to join a round in progress. This addition is reflected in function `TestIfSAProposalNeeded`. The slow agreement algorithm terminates once there is agreement not only on the `NSView`, but also on the `propNum`. This change in the termination condition is reflected in the function `TestIfAgreeementReached`. In Figure 3 we show the complete pseudo-code for these functions as implemented in the combined algorithm. Code that is not part of the slow agreement algorithm is shown in gray.

5.4 The combined algorithm

The combined algorithm works as follows: The server initially is not running either algorithm. When a `NE` is received from the notification service, the server begins running the fast agreement algorithm. It sends a `proposal` message of `type FA` to the other servers, and waits to receive similar `proposal` messages from all of the servers.

When the server receives a `proposal` message that matches its `NSView`, if it is a `proposal` message with

```
TestIfSAProposalNeeded(proposal inProp)
   if ( running ≠ SA ) then                 // detect if FA round blocked
      return ( running = none ∨ inProp.usedProps[me] = propNum ∨ inProp.type = SA )
   else                                     // detect if later SA round in progress
      return ( propNum < inProp.propNum )
   endif

TestIfAgreeementReached()
   if ( running = FA ) then                 // FA check: all FA proposals received
      return ( ∀s ∈ serversOf(NSView) : props[s].members = NSView ∧ props[s].type = FA )
   else                                     // SA check: all same round SA proposals received
      return ( ∀s ∈ serversOf(NSView) : props[s].members = NSView ∧ props[s].type = SA ∧
                                        props[s].propNum = propNum )
   endif

SendSAProposal(proposal inProp)
   // Notify the clients that a membership change is starting
   startChangeNum = max( curView.id, startChangeNum + 1 )
   send startChange ⟨startChangeNum, NSView⟩ to local(NSView)
   running = SA
   if ( inProp.type = FA ) then             // detected FA problem – initiate SA (new round)
      propNum = max( propNum + 1, props[serversOf(NSView)].propNum )
   else                                     // received SA proposal – join SA (same round)
      propNum = max( propNum, props[serversOf(NSView)].propNum )
   endif
   proposal outProp = ⟨me, NSView, startChangeNum, SA, usedProps[serversOf(NSView)], propNum⟩
   send outProp to serversOf(NSView) \ {me}
   deliver outProp immediately to myself // Invoke proposal handler
```

Code shown in gray is not part of the slow agreement algorithm.

Figure 3: Function definitions for the membership algorithm.

`type SA` it joins the slow agreement algorithm. If it is a `proposal` message with `type FA`, it runs the detection mechanism to check if the slow agreement algorithm needs to be started. In either of these cases, if the slow agreement algorithm is begun, the server sends a `proposal` message of `type SA`.

While the server is running either agreement algorithm, it waits to collect `proposal` messages from the other servers, until it has the necessary set to send a `view` as per the current (fast or slow) agreement algorithm. When a `view` is sent, the server returns to not running either algorithm.

If the server receives a new `NE` while running the slow agreement algorithm, it begins the fast agreement algorithm anew to avoid sending an obsolete view to the clients.

6 Performance Results

We now describe some performance measurements of our algorithm. The algorithm was run by machines at two locations, MIT and UCSD, communicating over the Internet. In each test, we ran a series of 150 membership changes per configuration: 50 changes in which the membership grew, 50 in which the membership shrank, and 50 in which the network event notification service mistakenly thought that the membership was changing. The former 100 cases were resolved by the fast agreement algorithm, whereas the latter 50 were resolved by the slow agreement algorithm.

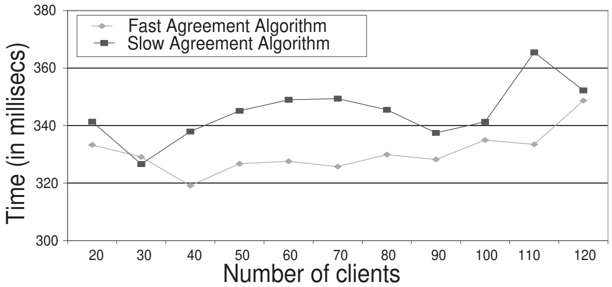

Figure 4: Client scalability (four servers).

In the first test we measured how the membership algorithm scales with respect to the number of clients per server. We held the number of servers constant (four servers - two at each location) and gradually increased the number of clients. Figure 4 shows the average observed time between clients initiating a membership change and the membership change being reported to the clients in this test. The measurements show that there appears to be a linear increase in the client response time with respect to the the number of clients, but the slope of this increase is quite small. Furthermore, the slow agreement algorithm is slower than the fast agreement algorithm, as expected. The difference between these two algorithms appears to range up to about 10% of the client response time for the fast agreement algorithm.

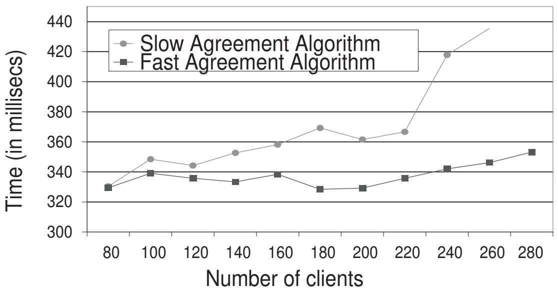

Figure 5: Server scalability.

In the second test, we measured how the membership algorithm scales with respect to the number of servers. We increased the number of servers in the protocol, and kept the number of clients per server constant at 20. The measurements are shown in Figure 5. Again we see that the client response time of the fast agreement algorithm is linear with a very small slope. For this test, the slow agreement algorithm appear to again be linear, but with a slightly larger slope. For more information on the performance measurements, see [21].

7 Discussion and Related work

We have described a scalable, one-round membership algorithm for wide-area networks. We have proven in [17] that this algorithm provides properties that are useful and attainable in an asynchronous system that may suffer communication partitions, but eventually stabilizes. We now compare our service with related work.

Following the approach taken by CONGRESS [4] and Maestro [7], our design separates the maintenance of membership from the group multicast: membership is not maintained by every client but only by dedicated membership servers that are not concerned with communication among clients. Our membership algorithm extends CONGRESS and provides an interface for virtually synchronous communication semantics [18]. Unlike Maestro [7], our membership service does not wait for responses from clients asserting that virtual synchrony was achieved before delivering views. Instead, we provide a novel interface that allows clients to implement virtual synchrony in parallel with the membership's agreement on views, and yet does not slow the agreement on views until responses from clients are received.

Other group communication systems that were designed for use in a WAN evolved from previous work on group communication systems for use in a LAN [10, 2, 3]. These systems leverage the idea that WANs are interconnected LANs. They first run the original algorithm in each LAN, and then run another algorithm among the LANs, merging the individual LAN memberships into one membership which is then disseminated to all of the group members. Thus, these algorithms overcome the problem of remote failure detection by having the failure detection done at the LAN level. However, these algorithms are inherently multi-round, since an additional round is added to the algorithm run on each LAN. For example, the Totem multiple ring algorithm [2] takes two rounds per ring[3] plus an extra round for multiple rings [2]. Furthermore, ours is the only membership algorithm that we are aware of that never delivers views which it knows to be obsolete. As explained in Section 2.1, this feature is important in WANs.

Light-weight group membership services (e.g., [10, 3, 20, 15]) employ a client-server approach to both virtual synchrony and membership maintenance. In these algorithms, there are two levels of membership, *heavy-weight* and *light-weight*. The servers are typically part of the heavy-weight membership, and they use virtually synchronous communication among them. The clients are typically part of the light-weight membership. Most light-weight group membership services, e.g., [10, 3, 20, 15], do not preserve the semantics of the underlying heavy-weight membership services. Unlike light-weight group membership algorithms, which compute both heavy-weight and light-weight membership, our algorithm only computes the process-level group membership, hence additional message rounds for computing both memberships are not necessary. Furthermore, our service provides clients with full virtual synchrony semantics.

The only other single round membership algorithm that we are aware of is the one-round membership algorithm in [9]. This algorithm terminates within one

[3]A ring is the logical representation of a LAN in Totem.

round in case of a single process crash or join, but in case of network events that affect multiple processes, the algorithm may take a linear number of rounds, where in each round a token revolves around a virtual ring consisting of all of the processes in the system. Thus, the latency until the membership is complete and stable is $O(n^2\delta)$ where δ is the maximum message delay at stable times. Thus, this membership algorithm is not suitable for WANs, where δ tends to be big and typical network events are partitions and merges. Once the network stabilizes and all of the information about network events has been propagated by the notification service to all of the servers, our algorithm terminates within at most 3δ time.

Acknowledgments

We are thankful to Tal Anker, Gregory Chockler, Roger Khazan and Ohad Rodeh for many interesting discussions and helpful suggestions.

References

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

[2] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The Totem Multiple-Ring Ordering and Topology Maintenance Protocol. *ACM transactions on computer systems*, 16(2), May 1998.

[3] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. TR CNDS-98-4, Johns Hopkins University, 1998.

[4] T. Anker, D. Breitgand, D. Dolev, and Z. Levy. CONGRESS: Connection-oriented group-address resolution service. In *Proceedings of SPIE on Broadband Networking Technologies*, November 2-3 1997.

[5] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In *Networks in Distributed Computing (DIMACS workshop)*, volume 45 of *DIMACS*, pages 23–42. AMS, 1998.

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

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

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

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

[10] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4), April 1996.

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

[12] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

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

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

[15] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Lightweight process groups in the Isis system. *Distributed Systems Engineering*, 1:29–36, 1993.

[16] M. Hayden. *The Ensemble System*. Phd thesis, Cornell University, Computer Science, 1998.

[17] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. Tech. Report CS99-623, Dept. of Computer Science and Engineering, University of California, San Diego, June 1999.

[18] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20th International Conference on Distributed Computing Systems (ICDCS)*, April 2000.

[19] 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.

[20] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer Verlag, 1991.

[21] J. Sussman. *Group Communication Services versus Wide-Area Networks*. PhD thesis, University of California, San Diego, 1999.

[22] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. TR 94-1442, dept. of Computer Science, Cornell University, August 1994.

[23] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A Gossip-Style Failure Detection Service. TR TR98-1687, Cornell University, Computer Science, May 1998.

[24] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report MIT-LCS-TR-790, MIT, Lab for Computer Science.