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

Idit Keidar
Lab for Computer Science
Massachusetts Institute of Technology
idish@theory.lcs.mit.edu
http://theory.lcs.mit.edu/~idish

Jeremy Sussman
Department of Computer Science and Engineering
University of California, San Diego
jsussman@cs.ucsd.edu
http://www.cs.ucsd.edu/~jsussman

Keith Marzullo
Department of Computer Science and Engineering
University of California, San Diego
marzullo@cs.ucsd.edu
http://www.cs.ucsd.edu/~marzullo

Danny Dolev
Computer Science Institute
The Hebrew University of Jerusalem, Israel
dolev@cs.huji.ac.il
http://www.cs.huji.ac.il/~dolev

Massachusetts Institute of Technology
Technical Memorandum MIT-LCS-TM-593

University of California, San Diego
Technical Report CS99-623

June 1999

Abstract

We describe a novel scalable group membership algorithm designed for wide area networks (WANs). Our membership service does not evolve from existing LAN-oriented membership services; it was designed explicitly for WANs. Our algorithm provides agreement on membership in a single message round in most cases, yielding a low message overhead. It avoids flooding the network and uses a scalable failure detection service designed for WANs. Furthermore, our algorithm avoids notifying the application of obsolete membership views when the network is unstable, and yet it converges when the network stabilizes.

In contrast to most group membership services, we separate membership maintenance from reliable communication in multicast groups: membership is not maintained by every process, but only by dedicated servers. The membership servers are not involved in the communication among the members of the groups. This design makes our membership service scalable in the number of groups supported, in the number of members in each group, and in the topology spanned by each group. Our service is complemented by a virtually synchronous communication service which provides clients with full virtual synchrony semantics.

1 Introduction

Group communication is a means of arranging processes into multicast groups. Group communication [1] 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.

Group membership is an example of a *distributed coordination* problem [44]. 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., [3, 6, 35, 29, 49, 13, 11, 36, 37]). Such applications greatly benefit from *virtually synchronous* communication semantics [17, 40, 30, 27, 52] 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. 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. Issues that need to be addressed include:

- Message latency tends to be large and highly unpredictable in a WAN, as compared to the relative consistency of message latency in a local-area network (LAN). This high latency works against algorithms in which processes repeatedly exchange messages in order to reach a decision.

- Failure detection in a WAN is usually less accurate than failure detection in a LAN. Inaccurate failure detection may cause a membership algorithm to change views frequently. Frequent view changes are costly as they can cause applications to engage in additional communication for re-synchronizing their shared state.

- There is no efficient support for the flooding of messages in a WAN, as opposed to a LAN. A group membership service supporting multiple groups in a wide area network must take care not to flood the network.

In this paper, we present a group membership algorithm which is designed for supporting hundreds of clients in WANs. We designed our server with a "fresh" approach: in contrast to previously suggested WAN-oriented group membership services, our server does not evolve from LAN-oriented membership algorithms. Rather it is designed explicitly with WAN considerations in mind.

Our membership algorithm addresses the challenges listed above. First, it minimizes message exchange in the common case where the failure detection is relatively consistent. View changes occur less frequently by using a failure detection mechanism better suited to WANs and by avoiding the delivery of *obsolete* views, which are views that reflect a membership that is already known to be out of date. Finally, it avoids flooding the network by propagating membership updates only to those who need them, and by using a client-server design in which the membership is not maintained by every process, but only by dedicated membership servers. These features are further explained in Section 2.

Our membership algorithm is implemented as part of a novel group membership service [10] designed for *computer supported cooperative work (CSCW)* [45] applications in WANs. Our algorithm is complemented by a virtually synchronous communication service. Our membership algorithm is *partitionable* [27, 52, 15], i.e., allows several disjoint views to exist concurrently.

The rest of this paper is organized as follows: In Section 2 we discuss the key features of our membership algorithm. In Section 3 we describe the environment model. In Section 4 we specify the guarantees of our membership service. In Section 5 we present the algorithm for maintaining group membership, and in Section 6 we prove the algorithm's correctness. In Section 7 we briefly describe how clients may implement virtual synchrony in conjunction with our service. Section 8 contains a concluding discussion of our work and comparison with related work.

2 The main features of the membership algorithm

We now discuss the key features of the membership algorithm presented herein.

2.1 Avoiding delivery of obsolete views

During the period in which the membership service is attempting to come to agreement on a view, further changes in the network connectivity can occur. Such concurrent changes are more likely due to the inaccuracy of failure detection combined with the fact that in WANs membership may be highly dynamic. Existing group membership algorithms [24, 26, 41, 50, 30] 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.

Unfortunately, membership changes cause extra overhead for applications to process. They can cause severe execution penalties to primary-backup applications (e.g., replicated databases), where a view change can initiate a lengthy recovery process in order to fail-over to a new primary. Furthermore, view changes typically cause applications to send special messages in order to re-synchronize their shared state (for examples, please see the applications in [4, 49, 29, 35, 6, 13, 11]). 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.

Note that avoiding the delivery of obsolete views implies that our membership algorithm may be non-terminating if the network does not stabilize, i.e., if the network situation constantly changes. However, if the network does stabilize, then our algorithm does terminate and does not initiate new membership changes unless new network events occur. We make this property formal in Section 4. Note that unstable networks force a membership service to either constantly deliver new views or else deliver none; we believe that in such situations it is better not to deliver any views. This avoids network congestion due to extra view change notifications. Furthermore, messages sent in an obsolete view cannot become *stable* (*safe*), i.e., they cannot be guaranteed to be delivered by all of the members of a view. Many applications (e.g., [35, 29, 6, 36, 37]) wait for messages to become stable before they act upon them. Thus, obsolete views increase network congestion by withholding information from applications that might allow them to avoid sending messages that will be discarded.

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 message rounds exchanged among the servers. In most cases, our algorithm provides agreement on the new view in a single communication round among the servers: once a change in network connectivity is detected, each server multicasts a message to the other servers and these messages are used in an agreement algorithm. Thus, if the maximum message latency in the network is δ, 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, 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 Using a network event notification service designed for WANs

Group membership services respond to network events (e.g., process crashes, communication link failures and recoveries) and to requests by a process to join or leave a certain multicast group. To this end, group membership algorithms use a network event notification (or failure detection) mechanism that informs them of network events. Typically, group communication systems implement such a mechanism using time-outs [26, 21, 41]. Unfortunately, detecting faults by setting timeouts on remote processes in a WAN is bound to be inaccurate since message latency in a WAN tends to be large and highly unpredictable.

Our membership service does not explicitly attempt to detect failures using time-outs. Instead, it uses a *network event notification service* as a building block. In our implementation, we use CONGRESS [9, 8] which is a distributed network event notification service geared to WANs. For example, CONGRESS servers use time-outs only on neighboring servers and local clients, rather than on processes that are several hops away. CONGRESS servers propagate information about network events and about a process joining or leaving a group voluntarily. In addition to time-outs, CONGRESS uses other techniques to detect failures where appropriate (see e.g., [53]). Furthermore, CONGRESS avoids flooding the network. Although in our implementation we use CONGRESS, it is worth noting that our algorithm may use any other similar service, e.g., the gossip-based failure detector of [51].

2.4 Avoiding flooding

A typical multicast group over a WAN may consist of a large number of members, which may be geographically spread far apart. A group membership service may need to maintain a large number of such groups. These conditions cause membership to be highly dynamic. An algorithm that manages the membership information will be forced to propagate large amounts of membership data across long distances. It is important not to flood the network in such a setting, and to propagate information only to those who need it. Both our membership algorithm and the underlying network event notification service we use (namely, CONGRESS) propagate information related to a certain group's membership only among the servers which have clients in this group. The servers forward views to their local clients who are members of this group.

2.5 A client-server design

Our membership algorithm is part of a novel architecture for group membership services designed for CSCW [45] and groupware applications in WANs. This architecture employs a client-server approach: dedicated membership servers maintain process-level group membership (i.e., which clients are members of each group). The servers do not explicitly maintain the server-level group membership. 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. It also allows an application to choose, on a per message stream basis, whether to use CONGRESS or CONGRESS augmented with membership semantics. Details on this architecture and its utility can be found in [10].

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. The membership server interface is presented in Section 4. In Section 7 we explain how the clients can use this interface to provide virtual synchrony.

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.

Our algorithm exploits two underlying services: It learns about the status of processes and links via the network event notification service, described in Section 3.1; and it exploits a reliable FIFO communication layer that operates in conjunction with the notification service, so that if a message is sent from one process to another then either this message eventually arrives or else the notification service reports the link to be faulty. This guarantee is made formal in Section 3.2.

3.1 The underlying network event notification service

Our membership service exploits 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 multicast groups.

Clients use the notification service in order to request to join or leave groups. The services are provided to clients by an interface that consists of the following basic functions:

join(G) is a request to make the invoking client a member of group G.

leave(G) is a request to remove the invoking member from the membership of G.

Our membership servers extend the notification service: each membership server has a local notification service component which 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(Group G, Set joining, Set leaving) is a notification that the processes in the set `joining` are joining group G, and those in the set `leaving` are either leaving the group or are suspected of having crashed or detached.

An `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 `NE`s that correspond to G as follows: The `NSView` is initially empty, and every time an `NE` arrives, the `NSView` becomes: $\texttt{NSView} \cup \texttt{NE.joining} \setminus \texttt{NE.leaving}$. Note that the `NSView` is not a membership view, since it has no unique identifier which can be agreed upon. The `NSView` is simply the list of group members that are currently not suspected.

Exploiting a notification service in a membership algorithm is conceptually no different than exploiting an underlying failure detector module, as done in practically all group membership algorithms, either explicitly [26, 15, 14, 39] or implicitly by using time-outs [21, 41]. We decouple the notification service from our membership algorithm in order to allow for efficient WAN implementations of the notification service.

As a failure detector in an asynchronous environment, the notification service is bound to be unreliable in some runs [20]: 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, as implied by Property 3.2 below.

3.2 The guarantees of the underlying communication

The reliable FIFO communication layer guarantees that messages from a single source are not received out of order. Formally:

Property 3.1 (FIFO Order) *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 as follows:

Property 3.2 (Reliable Links) *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 [48] as long as they are not suspected by the notification service. Group membership algorithms are often based on such services, e.g., in Transis [27, 26], Ensemble [34] and the algorithm described in [14, 15].

4 The guarantees of the membership algorithm

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 which contain a membership and a unique identifier. Each membership server communicates with its clients using reliable FIFO links. The client-server interaction is summarized in Figure 1.

The server sends two types of events to its clients:

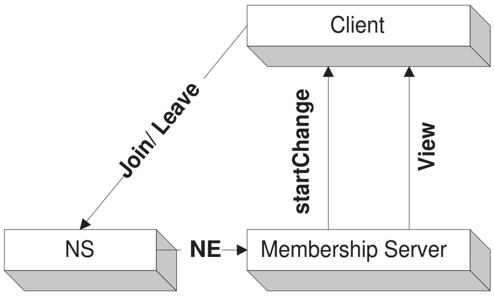

Figure 1: The membership service client-server interface.

`startChange`$(G, startChangeNum)$ indicates to the client that the server is now engaging in a membership change for group G. The `startChange` message is a hook that provides the synchronization needed by the clients to implement virtual synchrony in parallel with the membership's agreement on the view, as explained in Section 7.

`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, `members` is a set of processes and `startChangeNums` is a function from the servers of `members` to identifiers that were sent to the clients in `startChange` messages. This function is used in the implementation of virtual synchrony, as described in Section 7.

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:

Property 4.1 (View Identifier Local Monotonicity) *If a process delivers a view $V1$ and later delivers a view $V2$, then $V2.id > V1.id$.*

As explained above, 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 [52, 19]. An unstable communication layer can force every deterministic membership algorithm to either block or to constantly deliver changing views.

Therefore, we formulate Property 4.2 (Agreement on Views) below to guarantee only that agreement be reached in runs in which the network stabilizes and the failure detector module (or notification service) does not suspect correct and connected processes.

Property 4.2 (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, such that $V.group = G$ and $V.members = CS$, and do not receive new* `view` *or* `startChange` *messages in group G henceforward.*

Property 4.2 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 *eventual perfect* failure detector , i.e., it eventually stops making mistakes (eventual perfect failure detectors are discussed in [20, 22, 23, 14, 15]). A similar guarantee is formally defined in terms of network stability and failure detector properties in [14, 15, 52]. For the sake of simplicity, we have summarized both conditions into one requirement, namely that the servers eventually have the same `NSView`, and that this `NSView` does not change henceforward.

It is important to note that although Property 4.2 is guaranteed to hold only in certain runs, the conditions on these runs are *external* to the algorithm implementation, and therefore cannot be met by a trivial or useless algorithm.

Note also that we require stability to last forever. In practice, however, it only has to hold long enough for the membership algorithm to execute and for the failure detector module to stabilize, as explained in [28, 32]. This time period depends on external conditions: message latency, process scheduling and processing time. In this paper, we argue that the 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 [29, 21], by explicitly linking the guarantees of the membership algorithm to pre-determined bounds on process scheduling times and network delays.

4.2 Client Interface Guarantees

The `startChange` messages and `startChangeNums` are used by the clients for implementing virtual synchrony. In order to be useful they have to satisfy the following two properties:

Property 4.3 (Monotonicity of startChange Identifiers) *The* `startChange` *identifiers sent to each client are monotonically increasing.*

Property 4.4 (Integrity of startChange Identifiers) *Each* `view` *message V sent to a client c by a membership server s is preceded by a* `startChange` *message SM such that no messages are sent from s to c between SM and V, and V.*`startChangeNums[s]`$= SM.startChangeNum$.

5 The membership algorithm

In this section we discuss the implementation of the group membership algorithm. For the sake of simplicity, in this section 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: Once a server receives a `NE` from the notification service, the server notifies its clients that the membership is undergoing a change via `startChange` messages. At the same time, the server multicasts a `proposal` message to all of the other servers so the

[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.

servers can agree on the unique identifier of the view to be installed in a manner consistent with Property 4.1. Once the server has received a `proposal` message from each of the servers, the server computes the new view identifier and sends a `view` message to its clients. An example of this message flow, resulting from a client joining the group, is illustrated in Figure 2.

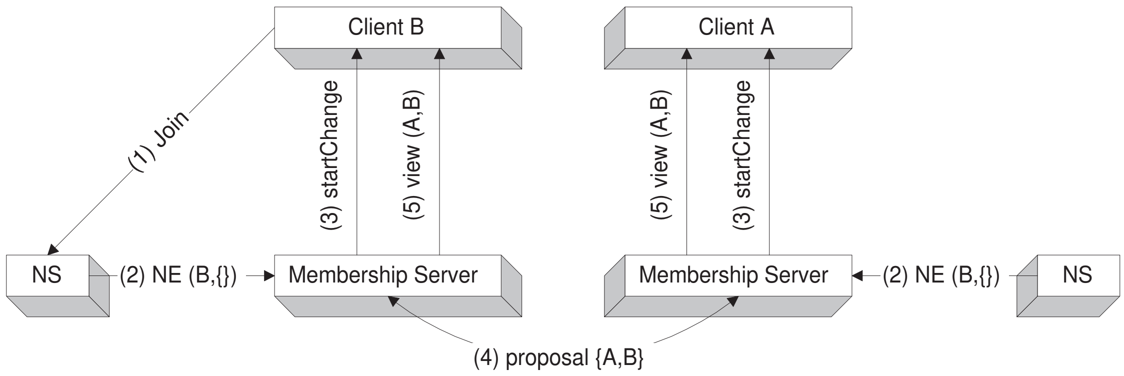

Figure 2: The membership service typical message flow.

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' which 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* that terminates in all cases.

The rest of this section presents the membership algorithm in pseudo-code. We begin in Section 5.1 by presenting the pseudo-code which is run in the typical case, i.e., the fast agreement algorithm. In Section 5.2 we describe a mechanism for detecting when this algorithm blocks. In Section 5.3 we describe the slow agreement algorithm. Finally, in Section 5.4 we put all of the pieces together and describe how the combined algorithm works.

5.1 The Fast Agreement Algorithm

Variables and types

The following message types are sent in the algorithm: Servers send each other `proposal` messages, and send clients `startChange` and `view` messages. The types and variables used by the membership algorithm are shown in Figure 3. The variables that are not used in the fast agreement algorithm are shown in gray.

The variable `running` is used to track which algorithm is currently being run: it has the value `none` if no algorithm is being run, the value `FA` if the fast agreement algorithm is being run, and the value `SA` if the slow agreement algorithm is being run. The variable `NSView` contains the aggregation

```
Type CSet SetOf(clients)
Type NE  ⟨Set joining, Set leaving⟩
Type algType {none, FA, SA}

Message types:
   S→C view ≜ ⟨int id, CSet members, startChangeNums[serversOf(members) ↦ int]⟩
   S→C startChange ≜ ⟨int startChangeNum⟩
   S→S proposal ≜ ⟨sender, CSet members, int startChangeNum,
                     algType type, usedProps[servers ↦ int], int propNum ⟩

Variables and Data Structures:
   serverId     me                                   // My server name
   algType      running = none                       // Initially not running any algorithm
   CSet         NSView = { }                         // aggregation of all NEs
   function     props [servers ↦ proposal] = null    // last proposal received from server
   view         curView = ⟨0, { }, [* ↦ 0]⟩          // last view delivered
   int          startChangeNum = 0                   // unique id for startChange messages
   int          propNum = 0                          // proposal logical clock
   function     usedProps [servers ↦ int] = 0        // proposal from server used for view

Assumed external functions:
   serversOf[clients ↦  servers]                     // returns the servers of the clients
   local[CSet ↦ local clients]                       // returns local clients out of a set of clients
```

Variables shown in gray are not part of the fast agreement algorithm.

Figure 3: Types and variables for the membership algorithm.

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. The variable `startChangeNum` is used to ensure that every `startChange` message sent to a client has a monotonically increasing number (as required in Section 4). The variable `propNum` is a logical timestamp used to ensure that every `proposal` message sent by a server has a unique monotonically increasing identifier. The variable `usedProps` is used to detect if the fast agreement algorithm has failed, 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 to the notification service, 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 simple 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 interested reader is referred to [42] for details.

```
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) 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 4: Event handlers for the membership algorithm.

Event handlers

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 of the membership algorithm are presented in Figure 4. Code shown in gray is not part of the fast agreement algorithm.

The fast agreement algorithm follows precisely the message flow described above in Figure 2. 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 which 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 Property 4.1 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. Thus, the servers must be able to calculate such an identifier. Each `startChange` message sent to a client has a unique integer, the `startChangeNum`, greater than or equal to the identifier of the last `view` sent to that client. Then, the `startChangeNum` 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. The `startChangeNum` values are also included in the `view` message, in order to allow clients to correlate `startChange` events with the view.

Reaching agreement on a view is determined via the `proposal` messages sent by all of the servers of clients in the `NSView`. The `props` buffer is used to collect these `proposal` messages. Whenever a `proposal` message is received, it is placed in the `props` buffer regardless of the membership it proposes. Due to the FIFO nature of the communication (Property 3.1), this `proposal` message is guaranteed to have been sent after the `proposal` message that it replaces. By using the most recent `proposal` message sent by the servers, the algorithm avoids sending 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

In order to satisfy Property 4.2, 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*, since in such cases one or more servers will run the agreement protocol forever.

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

In this section we present a mechanism for detecting such cases. 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.

Thus, for the remainder of this section we assume the following: Let CS be a set of clients and SS be the set of servers which serve the clients in CS. We assume there is a time t_0 after which the `NSView` of every server in SS is and remains CS. We show that under this assumption, our detection mechanism will detect the need to invoke the slow agreement algorithm if and only if the fast agreement algorithm will block.

By time t_0, every server in SS has received its last `NE` from the notification service, and this `NE` makes the server's `NSView` $= CS$. Therefore, every server in SS will send a `proposal` message with `members` $= CS$ as its last `proposal` message. We use $last_s$ to refer to the last `proposal` message sent by a server s. For every server $s \in SS$, $last_s$ will be received by every server $s' \in SS$, according to Property 3.2.

If the `props` buffer of every server in SS contains the same set of `proposal` messages before sending a `view` to the clients, then the fast agreement algorithm terminates successfully – all of the servers in SS agree on the view, and all of the clients receive the exact same `view` message. Thus, the only way the fast agreement algorithm can fail is if there is some pair of servers $s, s' \in SS$, such that s does not use $last_s$ and $last_{s'}$ together for a view.

However, server s must receive $last_{s'}$, as explained above. Furthermore, since s' has clients in the `NSView` of s, s must use some `proposal` message from s' for the same view as $last_s$ unless it receives no such `proposal` message. Thus, $last_{s'}$ is not used by s for the same view as $last_s$ in only two cases: 1. s uses some earlier `proposal` message from s' for the same view as $last_s$; or 2. s uses $last_{s'}$ for a view with an earlier `proposal` message of its own. We now explain the detection mechanism and prove that it detects both of these cases.

The detection mechanism is implemented in the function `TestIfSAProposalNeeded`, which is invoked whenever a `proposal` message *prop* is received by some server s, as shown in gray in the event handler of Figure 4. The detection mechanism is presented in Figure 5.

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

Figure 5: Detecting if the fast agreement algorithm is blocked.

The detection mechanism detects the two cases described above:

1. Case 1 is detected because $last_{s'}$ arrives after s already sent a view using $last_s$. Therefore when the `proposal` $last_{s'}$ arrives, the `running` variable at s is `none`, and s detects the block.

2. Case 2 is detected by s' using the function `usedProps` included in the `proposal` $last_s$. (The code for maintaining `usedProps` is shown in gray in Figure 4). $last_s$.`usedProps[s']` contains the `propNum` of the latest `proposal` from s' which was used for a view by s. Thus, if s used $last_{s'}$ for a view before sending $last_s$, then $last_s$.`usedProps[s']` is equal to $last_{s'}$.`propNum`. When $last_s$ reaches s', the value of `propNum` at s' is equal to $last_{s'}$.`propNum` (since propNum is increased only when a `proposal` is being sent). Thus, $last_s$.`usedProps[s']` is equal to the value of `propNum` at s' and s' detects the block.

In Lemma 6.4 we formally prove that whenever the fast agreement algorithm blocks, it is detected by the detection mechanism at some server. Furthermore, in Lemma 6.5 we 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. In order to agree on a common round number, another level of knowledge is required.

The slow agreement algorithm is begun by a server when it detects that the fast agreement protocol 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 or received. 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 which 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.

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 protocol, 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.

Slow agreement pseudo-code

In Figure 6, we complete the pseudo-code shown in Figure 4 by adding the functions which 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 4).

The function `SendSAProposal` is also used by the slow agreement protocol 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 6 we show the complete pseudo-code for these functions as implemented in the combined algorithm. Code which is not part of the slow agreement algorithm is shown in gray.

[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.

```
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) 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 6: Function definitions for the membership algorithm.

5.4 Putting the pieces together: 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 which matches its `NSView`, if it is a `proposal` message with `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` from the notification service while running the slow agreement algorithm, it begins the fast agreement algorithm anew, sending a `proposal` message of `type FA`. It is this mechanism by which the algorithm avoids sending obsolete views to the clients.

The combined algorithm can be represented as a state machine with three states – a state in

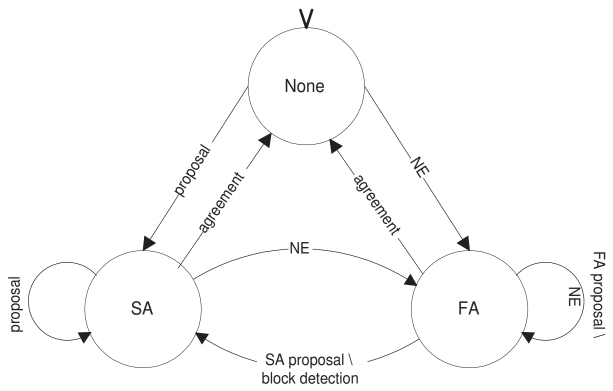

Figure 7: The membership algorithm state diagram.

which the server is running the fast agreement algorithm, a state in which the server is running the slow agreement algorithm, and a state in which the server is running neither algorithm. This state machine is presented in Figure 7.

6 Correctness of the membership algorithm

We now prove that the algorithm fulfills the properties specified in Section 4. In Section 6.1 we prove that it fulfills the client interface properties: 4.3 and 4.4. In Section 6.2, we prove that it fulfills the membership properties: 4.1 and 4.2.

6.1 Client interface properties

Lemma 6.1 *(Monotonicity of startChange Identifiers) Property 4.3 holds, i.e.,* `startChange` *identifiers sent to each client are monotonically increasing.*

Proof: Whenever a `startChange` message is sent to the clients, `startChangeNum` is first increased and then sent in the message. ■

Lemma 6.2 *(Integrity of startChange Identifiers)*
Property 4.4 holds, i.e., each `view` *message V sent to a client c by a membership server s is preceded by a* `startChange` *message SM such that no messages are sent from s to c between SM and V, and V.*`startChangeNums[s]`$= SM.startChangeNum$.

Proof: A server s sends its clients two types of messages: `view` and `startChange`. Whenever a `startChange` message is sent, the server also sends a `proposal` which includes the latest `startChange.startChangeNum` sent to its clients, and invokes the proposal handler which stores this `proposal` in `props[s]`. Before sending a `view` message, the server checks that `props` contains `proposal` messages from all of the servers of members of the view, including itself. `view.startChangeNums[s]` is then selected to be `props[s].startChangeNum`, which contains the latest `startChange.startChangeNum` sent to local members of the view.

Upon sending a `view`, the server removes this `proposal` messages from `props`. Therefore, each `view` must be preceded by a sending of a `proposal` message which follows the previous view. Moreover, every time a `proposal` is sent, `startChange` messages are sent to all of the clients who are members of the proposed view. ■

6.2 Membership properties

Lemma 6.3 *(Local Monotonicity) Property 4.1 holds, i.e., if a client receives a view $V1$ and later receives a view $V2$, then $V2.id > V1.id$.*

Proof: Whenever a `view` V is sent, $V.id$ is chosen to be greater than the $startChangeNum$ of the last `startChange` sent to local clients. Whenever a `startChange` message is sent to local clients, $startChangeNum$ is chosen to be greater than `curView.id`. The proof follows from Lemma 6.2. ■

Let CS be a set of clients, and SS the set of servers serving clients in CS. For the rest of this section we assume that there is a time t_0 such that from time t_0 onwards, the `NSView` at all of the servers in SS contains exactly the clients in CS.

Lemma 6.4 *(Fast Agreement Blocking Detection) If the fast agreement algorithm does not terminate successfully, then the detection mechanism (described in Figure 5) detects the blocking after time t_0.*

Proof: If every server sends a view to its clients based on the last `proposal` message sent by the servers in SS, then the fast agreement algorithm terminates successfully. Therefore, the fast agreement algorithm fails to terminate in only two cases:

1. There exists some pair of servers $s, s' \in SS$ such that for some view V, s uses some earlier `proposal` message from s' with $last_s$.

2. There exists some pair of servers $s, s' \in SS$ such that for some view V, s uses $last_{s'}$ with an earlier `proposal` message of its own.

We now prove that both of these cases will result in detections, i.e., `TestIfSAProposalNeeded` at one of the servers will return TRUE.

In the first case, for view V, s uses $last_s$ and a `proposal` message $p_{s'}$ from s' that precedes $last_{s'}$. After using $last_s$, s receives no further `NEs` from the notification service. Thus, s does not run the fast agreement algorithm again so `running` at s will remain `none` after it sends V. Due to the FIFO nature of the links described in Property 3.1, all `proposal` messages from s' received by s are received in the order they are sent. Thus, $last_{s'}$ will be received by s after $p_{s'}$. Since s uses $p_{s'}$ for view V, $last_{s'}$ is received by s after it sends V. Thus, when s receives $last_{s'}$ `running` will be `none`, and this will result in detection at s.

In the second case, for view V, s uses $last_{s'}$ and a `proposal` message p_s that s sent before sending $last_s$. If s' also uses $last_{s'}$ with some `proposal` message that s sent before sending $last_s$ for some view V', then s' will detect the failure, as described in the first case above. So the case we are examining is reduced to s using $last_{s'}$ and p_s for view V while s' does not send a `view` using $last_{s'}$ and any earlier `proposal` message from s.

When s uses $last_{s'}$ and p_s for view V, s sets `usedProps[s']` to the `propNum` of $last_{s'}$. s always uses its most recent `proposal` message for a view. Therefore, s cannot have sent $last_s$ before it used $last_{s'}$. Thus, when s sends $last_s$, the value of `usedProps[s']` is the `propNum` of $last_{s'}$. Furthermore, s' must receive $last_s$ after it has already sent $last_{s'}$. Since, by assumption, s' will not use $last_{s'}$ with any earlier `proposal` message from s, $last_{s'}$ must still be in the `props` buffer of s' when s' receives $last_s$. Thus, the value `usedProps[s']` in $last_s$ will be equal to the `propNum` at s' when $last_s$ is received by s'. This will result in detection at s'. ■

Lemma 6.5 *(No False Blocking Detection) The detection mechanism described in Figure 5 detects blocking after time t_0 only if the fast agreement algorithm does not terminate successfully.*

Proof: We now prove that a detection will not occur if the fast agreement algorithm terminates successfully, i.e., `TestIfSAProposalNeeded` will not return TRUE at any server after time t_0.

If the fast agreement algorithm terminates successfully after time t_0, then every server s will send a view V using $last_{s'}$ for every s' in SS. Before s sends $last_s$, the `NSView` at s will not be CS, so a $last_{s'}$ received by s before it sends $last_s$ will not result in detection. By the time s sends V, s must have received $last_{s'}$ for every $s' \in SS$, by assumption. Therefore, s will not receive any further `proposal` messages from s' that might lead to a detection. Since `running` is set to `FA` from the time that s sends $last_s$ until it sends V, a detection will only occur if there is some $last_{s'}$ which has `usedProps[s]` set to the `propNum` of $last_s$.

The `usedProps` function of s' is updated before s' sends a `view` to its clients. At that time, `usedProps[s]` is set to the `proposal` used by s' for that view. By assumption, s' uses $last_s$ for the same view that it uses $last_{s'}$. Therefore, `usedProps[s]` at s' is not set to the `propNum` of $last_s$ until after $last_{s'}$ is sent. Thus no detection will occur if the fast agreement algorithm terminates correctly. ■

Lemma 6.6 *(Slow Agreement Termination) After time t_0, if a the slow agreement algorithm is started by some server s then there is some server $s' \in SS$ such that the slow agreement algorithm started by s' terminates at all servers.*

Proof: First, note that if the slow agreement protocol is invoked after time t_0 by some server s in SS, then eventually every server s' in SS will enter the slow agreement protocol by sending a `proposal` of type `SA`. Also, this will occur after s' has received its final `NE` from the notification service.

Second, note that any `proposal` sent in the slow agreement protocol by a server s has a greater `propNum` than any `proposal` of type `SA` received by s beforehand.

Third, note that `propNum` at server s is increased above the `propNum` of those `proposal` messages received by s only in response to a `NE` or upon reception of a `proposal` of type `FA`, and `proposal` messages of type `FA` are sent only in response to a `NE`. Since after time t_0 no `NE` is received by a server, there is a time $t_1 > t_0$ after which no more `proposal` messages of type `FA` are sent or received and thus `propNum` at s no longer increases above the `propNum` of other `proposal` messages.

Let n be the largest `propNum` which was sent in a `proposal` of type `SA`. By the argument above, if some server sends a `proposal` of type `SA` after t_0, then any server that sends a `proposal` of type `SA` with `propNum`$= n$ does so after time t_0. Therefore, from Property 3.2, all of the servers in SS receive this `proposal`, and all respond by sending `proposal` messages of type `SA` with `propNum`$= n$ (unless they have already done so). These `proposal` messages will also be received by all of the servers in SS. Furthermore, in all of these `proposal` messages, `NSView` is CS. Since no `proposal` messages of type `FA` and no `proposal` messages of type `SA` with a higher `propNum` will be sent, the slow agreement algorithm will terminate once all of these `proposal` messages are received. ■

Theorem 6.7 (Agreement on Views) *Let CS be 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` *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, such that $V.members = CS$, and do not receive new views or* `startChange` *messages henceforward.*

Proof: When each server receives the last `NE` from the notification service that sets its `NSView` to CS, it runs the fast agreement algorithm. If this agreement terminates successfully, all of the clients in CS will receive the same view. If it fails, then by Lemma 6.4, the slow agreement algorithm will be run. The slow agreement algorithm always terminates, as proven in Lemma 6.6.

What remains to be proven is that the clients will not receive any further `startChange` or `view` messages after that `view` is received. Due to the FIFO nature of communication, the clients will not receive a message from the server after the `view` unless the server sends another message.

The server only sends messages to the client if it begins or ends either of the agreement algorithms. Since the fast agreement algorithm in which we are interested is running after the last `NE` received by each server, the fast agreement algorithm will not be run again. If the slow agreement algorithm is run, and it terminates, then every server will have run the same round of the slow agreement algorithm and received all of the `proposal` messages, as described in Lemma 6.6. Thus, unless a stimulus to run another round of the slow agreement algorithm is received by some server, the slow agreement algorithm will not run again. But, the only stimulus to run this algorithm is from a detection that the fast agreement algorithm is blocked. Lemma 6.5 shows that the detection mechanism detects blocking after time t_0 only if the fast agreement algorithm does not terminate successfully. Thus, unless the fast agreement is run again, there will not be another run of the slow agreement algorithm. But, we have already argued that the fast agreement algorithm will not be run again. ∎

7 Providing virtual synchrony

Our group membership system is designed to be used in conjunction with a multicast service as part of a group communication system. Group communication systems generally provide some variant of virtual synchrony semantics; many such variants have been suggested [17, 40, 30, 52, 18, 47, 29]. While detailed discussion of all of these variants is beyond the scope of this paper, we describe here the most common properties of virtual synchrony and how clients can implement them in conjunction with our membership service.

The key aspect of virtual synchrony semantics is the interleaving of send and delivery events with views. In this model, send and delivery events of messages occur in views. We say that a multicast event e in group G occurs at process p in view V if V was the latest view that p delivered in group G before e, or a default initial view V_0 if no view was delivered.

All of the variants of virtual synchrony ensure that a message m is delivered in the same view V by all processes that deliver m, and that m is not delivered in a view that is ordered before the view in which the message was sent. Some of these semantics (e.g., *strong virtual synchrony* [30]) strengthen this property to ensure that the view in which a message is delivered is the same view in which it was sent. Another useful property provided by nearly all variants of virtual synchrony is the agreement of the processes moving together from view $V1$ to view $V2$ on the set of messages delivered in $V1$.

Virtual synchrony properties are implemented by synchronizing participating processes while view changes are taking place (for examples, please see [30, 33, 41]). During long periods of time in which a view does not change, the messages sent can be delivered with minimal interference from the virtual synchrony algorithm. During view changes, the algorithm agrees upon the set of messages to be delivered before moving to the new view. In order to implement virtual synchrony, a client c sends every other client a `flush` message which signifies that c has stopped sending messages in the current view.

Our membership service provides hooks that the clients can use to implement virtual synchrony while the servers are agreeing upon the view. Upon receiving a `startChange` message from the server, each client sends a `flush` message to the other clients. The `flush` message is tagged with the `startChangeNum` of the `startChange` message, and also carries the information required in order to agree on the set of the messages to be delivered in the view that is now ending. If strong virtual synchrony semantics are desired, then the client sends no more messages after sending a `flush` until the next view is delivered.

When a client receives a `view` message V from its server, the client has to make sure to deliver the same set of messages as other clients before delivering V to its application. To this end, the client collects `flush` messages from all of the clients that continue with it from the current view to V. Clients use the information in the `flush` messages to determine the set of messages to be delivered in the current view. Clients delay the delivery of V to the application until these messages are delivered. The `startChangeNums` mapping in the view message serves to make sure that the same set of `flush` messages are used for the same view at all of the clients: for each client c, V.`startChangeNums[serverOf(c)]` is the identifier of the `flush` message to be used from c.

8 Discussion and Related work

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

8.1 Separating membership maintenance from multicast services

Following the approach taken by CONGRESS [8, 9], Maestro [16] and Caelum [12], our design separates the maintenance of membership from the actual group multicast: membership is not maintained by every client but only by dedicated membership servers which are not concerned with the actual communication among clients in the groups.

Our membership algorithm extends CONGRESS and provides an interface for virtually synchronous communication semantics. Unlike Maestro [16] and Caelum [12], 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 which 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.

8.2 Scalable group communication services for WANs

Existing group communication systems that were designed for use in a WAN evolved from previous work on group communication systems for use in a LAN [24, 38, 5, 41, 7]. These systems leverage the idea that all WANs are interconnected LANs. These systems first run the original algorithm in each LAN, and then run another algorithm among the LANs, merging the individual memberships into one membership. This merged membership 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 [41].

The idea of using a two-layer hierarchy to support scalable virtual synchrony was discussed in [33]. The algorithm presented in this paper deals solely with the communication issue, and not

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

the membership. Our work complements this work by implementing the membership needed by such a system.

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 very important in WANs.

8.3 Light-weight group membership services

"Light-weight" group membership algorithms [25, 5, 24, 7, 43, 31, 46, 16] 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 part of the heavy-weight membership, and they use virtually synchronous communication among them. The clients are part of the light-weight membership. Most light-weight group membership services, e.g., [25, 5, 24, 7, 43, 31], do not preserve the semantics of the underlying heavy-weight membership services.

In these systems, when there is a membership change, the servers first compute the heavy-weight membership, and then map it to several light-weight process groups. As with the approach taken by us, this approach is scalable in the number of clients, since the membership algorithm involves reaching agreement among the servers only. However, computing the light-weight group membership requires additional communication after the heavy-weight membership algorithm is complete.

Unlike light-weight group membership algorithms, our algorithm only computes the process-level group membership, hence additional message rounds for computing the light-weight membership are not necessary. Furthermore, our service provides clients with full virtual synchrony semantics.

Light-weight group membership services have another important advantage: they scale well in the number of groups maintained, since they maintain the membership for several groups at the same time. Since in our design the same membership servers maintain the membership of all of the groups, our servers can also handle membership changes concerning several groups at the same time. Indeed, our implementation of the algorithm [42] also possess this feature: if there are concurrent notifications concerning multiple groups, the membership server handles all of these groups together, and bundles the messages corresponding to different groups into a single message.

Thus, our algorithm provides the full semantics of heavy-weight group membership along with the scalability and flexibility of a light-weight group membership, all for the cost of a single communication round in the common case.

8.4 One round membership algorithms

The only other single round membership algorithm that we are aware of is the one-round algorithm in [21]. This algorithm terminates within one 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. CONGRESS, the notification service we use in our implementation, propagates network information along a spanning tree. The depth of the tree depends solely on the network topology, and does not depend on the number of clients (i.e., members) in each group.

Acknowledgments

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

References

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

[2] D. A. Agarwal. *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks.* PhD thesis, University of California, Santa Barbara, 1994.

[3] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *3rd International Workshop on Services in Distributed and Networked Environment (SDNE)*, pages 84–91, June 1996.

[4] Y. Amir, G. V. Chokler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 183–192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997. Full version available as Technical Report CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[5] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS)*, July 1992.

[6] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[7] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. TR CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.

[8] T. Anker, D. Breitgand, D. Dolev, and Z. Levy. CONGRESS: CONnection-oriented Group-address RESolution Service. Tech. Report CS96-23, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, December 1996. Available from: `http://www.cs.huji.ac.il/~transis`.

[9] 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.

[10] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In M. Mavronicolas, M. Merritt, and N. Shavit, editors, *Networks in Distributed Computing (DIMACS workshop)*, volume 45 of *DIMACS*, pages 23–42. American Mathematical Society, 1998.

[11] T. Anker, G. Chockler, I. Keidar, M. Rozman, and J. Wexler. Exploiting group communication for highly available video-on-demand services. In *Proceedings of the IEEE 13th International*

Conference on Advanced Science and Technology (ICAST 97) and the 2nd International Conference on Multimedia Information Systems (ICMIS 97), pages 265–270, April 1997.

[12] T. Anker, G. V. Chockler, D. Dolev, and I. Keidar. The Caelum toolkit for CSCW: The sky is the limit. In *The Third International Workshop on Next Generation Information Technologies and Systems(NGITS 97)*, pages 69–76, June 1997.

[13] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 244–252, June 1999.

[14] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems. TR UBLCS-95-18, Department of Conmputer Science, University of Bologna, November 1995.

[15] 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.

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

[17] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 123–138. ACM, Nov 1987.

[18] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[19] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322–330, May 1996.

[20] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[21] 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.

[22] D. Dolev, R. Friedman, I. Keidar, and D. Malki. Failure Detectors in Omission Failure Environments. TR 96-13, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, September 1996. Also Technical Report 96-1608, Department of Computer Science, Cornell University.

[23] D. Dolev, R. Friedman, I. Keidar, and D. Malki. Failure detectors in omission failure environments. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, page 286, August 1997. Brief announcement.

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

[25] D. Dolev and D. Malki. The design of the Transis system. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 83–98. Springer Verlag, 1995. LNCS 938.

[26] 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.

[27] D. Dolev, D. Malki, and H. R. Strong. A Framework for Partitionable Membership Service. TR 95-4, Institute of Computer Science, The Hebrew University of Jerusalem, March 1995.

[28] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[29] 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.

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

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

[32] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, Oct. 1997. IEEE Computer Society Press.

[33] K. Guo, W. Vogels, and R. van Renesse. Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication. In *7th ACM SIGOPS European Workshop*, September 1996.

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

[35] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[36] R. Khazan. Group communication as a base for a load-balancing, replicated data service. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1998.

[37] R. Khazan, A. Fekete, and N. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on DIStributed Computing (DISC)*, pages 258–272, Andros, Greece, September 1998.

[38] D. Malkhi, Y. Amir, D. Dolev, and S. Kramer. The Transis approach to high availability cluster communication. TR 94-14, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[39] C. Malloth and A. Schiper. View synchronous communication in large scale networks. In *2nd Open Workshop of the ESPRIT project BROADCAST (Number 6360)*, July 1995 (also available as a Technical Report Nr. 94/84 at Ecole Polytechnique Fédérale de Lausanne (Switzerland), October 1994).

[40] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994. Full version: technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[41] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Commun. ACM*, 39(4), April 1996.

[42] A. Nowersztern. MOSHE: Membership Object-oriented Service for Heterogeneous Environments. Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, January 1998. Available from: `http://www.cs.huji.ac.il/~transis`.

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

[44] A. Ricciardi. Dissecting distributed coordination. In *9th International Workshop on Distributed Algorithms (WDAG)*, pages 101–118, September 1995.

[45] T. Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353, 1991.

[46] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, and K. Birman. A dynamic light-weight group service. In *15th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 23–25, Oct. 1996. also Cornell University Technical Report, TR96-1611, August, 1996.

[47] A. Schiper and A. Ricciardi. Virtually synchronous communication based on a weak failure suspector. *Digest of Papers, FTCS-23*, pages 534–543, June 93.

[48] I. Shnaiderman. Implementation of Reliable Datagram Service in the LAN environment. Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, January 1999. Available from: `http://www.cs.huji.ac.il/~transis/publications.html`.

[49] J. Sussman and K. Marzullo. The *bancomat* problem: An example of resource allocation in a partitionable asynchronous system. In *12th International Symposium on DIStributed Computing (DISC)*, September 1998. Full version: Tech Report 98-570 University of California, San Diego Department of Computer Science and Engineering.

[50] 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.

[51] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Style Failure Detection Service. TR TR98-1687, Cornell University, Computer Science, May 1998.

[52] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication System Specifications: A Comprehensive Study. Technical report, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1999. In preparation.

[53] W. Vogels. World wide failures. In *ACM SIGOPS 1996 European Workshop*, September 1996.