Moshe: A Group Membership Service for WANs

Idit Keidar
The Technion and MIT,
Jeremy Sussman
IBM T. J. Watson Research Center,
Keith Marzullo
University of California, San Diego,
Danny Dolev
The Hebrew University of Jerusalem.

We present Moshe, a novel scalable group membership algorithm built specifically for use in wide area networks (WANs), which can suffer partitions. Moshe is designed with three new significant features that are important in this setting: it avoids delivering views that reflect out-of-date memberships; it requires a single round of messages in the common case; and it employs a client-server design for scalability. Furthermore, Moshe's interface supplies the hooks needed to provide clients with full virtual synchrony semantics. We have implemented Moshe on top of a network event mechanism also designed specifically for use in a WAN.

In addition to specifying the properties of the algorithm and proving that this specification is met, we provide empirical results of an implementation of Moshe running over the Internet. The empirical results justify the assumptions made by our design and exhibit good performance. In particular, Moshe terminates within a single communication round over 98% of the time. The experimental results also lead to interesting observations regarding the performance of membership algorithms over the Internet.

Categories and Subject Descriptors: C.2.4 [**Computer-communication networks**]: Distributed Systems; D.4.7 [**Operating systems**]: Organization and Design—*Distributed Systems*; C.2.1

Preprint of an article to appear in ACM Transactions on Computer Systems (TOCS), 2002. A preliminary version of this paper appeared in the Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS), April 2000, pp. 356–365.
This work was supported by Air Force Aerospace Research (OSR) grants F49620-00-1-0097 and F49620-00-1-0327, Nippon Telegraph and Telephone (NTT) grant MIT9904-12, NSF grants CCR-9909114 and EIA-9901592, and by UT Austin grant 97-0039 (subcontract to DARPA).
Name: Idit Keidar
Affiliation: The Technion Department of Electrical Engineering and MIT Lab for Computer Science.
Address: Technion - Israel Institute of Technology, Department of Electrical Engineering, Technion City, Haifa, 32000 Israel. E-mail: idish@ee.technion.ac.il.
Name: Jeremy Sussman
Affiliation: IBM T. J. Watson Research Center.
Address: 30 Saw Mill River Road, Hawthorne, NY 10532, USA. E-mail: jsussman@us.ibm.com.
Name: Keith Marzullo
Affiliation: University of California, San Diego, Department of Computer Science and Engineering.
Address: 9500 Gilman Drive, La Jolla, CA 92093, USA. E-mail: marzullo@cs.ucsd.edu.
Name: Danny Dolev
Affiliation: The Hebrew University of Jerusalem, School of Engineering and Computer Science.
Address: Givat Ram, Jerusalem, 91904 Israel. E-mail: dolev@cs.huji.ac.il.

[**Computer-communication networks**]: Network Architecture and Design—*Network communications*

General Terms: Reliability, Algorithms

Additional Key Words and Phrases: Group communication systems, group membership, wide area networks, partitionable group membership, virtual synchrony, view synchrony.

---

1. INTRODUCTION

Group communication is a means of providing multi-point to multi-point communication by organizing processes into groups. A *group* is a set of processes which are said to be *members* of the group. For example, a group can arise from users editing a shared file. Another group can arise from users playing an on-line game with each other, and another, from participants in a multi-media conference. A process becomes a group member by requesting to *join* the group; it can cease being a member by requesting to *leave* the group or by failing. Each group is associated with a logical name. Processes communicate with group members by sending a message targeted to the group name; the group communication service delivers the message to the group members.

View-oriented group communication systems [ACM 1996; Chockler et al. 2001; Birman 1996] provide membership and reliable multicast services. The *membership* of a group is a list of the currently active and connected processes in a group. The task of a *group membership service* is to track the membership of the group, as it evolves over time. 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. The membership service strives to deliver the same views to mutually connected group members. Reliable multicast services that deliver messages to the current view members complement the membership service.

View-oriented group communication systems are especially useful for constructing fault-tolerant applications that consistently maintain replicated state of some sort (examples include [Amir et al. 1994; Keidar and Dolev 1996; Fekete et al. 2001; Sussman and Marzullo 1998; Anker et al. 1999; Khazan et al. 1998; Friedman and Vaysburg 1997; Guerraoui and Schiper 1997b]). Such applications greatly benefit from *virtually synchronous* communication semantics (for example, [Moser et al. 1994; Friedman and van Renesse 1995; Chockler et al. 2001; Babaoğlu et al. 2001; Schiper and Ricciardi 1993]), that synchronize views with regular messages and thus simulate a "benign" world in which message delivery is reliable within the set of connected processes. (See [ACM 1996; Birman 1996; Chockler et al. 2001] for discussion of the utility of group communication systems and virtually synchronous semantics). 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 membership service for a wide area network (WAN) is a challenging task. Issues that need to be addressed include:

—*High latency:* 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). In addition, message loss, which is very rare in LANs, is quite common in WANs. Message loss leads to retransmissions, which delay messages even further. The high latency works against algorithms in which processes repeatedly exchange messages in order to reach a decision.

—*Frequent changes:* Connectivity changes are more likely in a WAN than in a LAN. In addition, failure detection in a WAN is usually less accurate than failure detection in a LAN. Connectivity changes and inaccurate failure detection may cause a membership algorithm to change views frequently. This is costly as it can cause applications to engage in additional communication for re-synchronizing their shared state.

—*Instability:* The status of communication paths in a WAN often fluctuates frequently due to link failures or congestion. Lack of transitivity is also not uncommon over the Internet: in our experiments we observed periods of up to half an hour during which communication was not transitive. We refer to periods with non-transitive communication or frequent connectivity changes as *unstable*. A group membership algorithm for WANs should be designed taking into account that unstable periods can occur and endure for significant periods.

In this paper, we present Moshe, a group membership algorithm to support virtually synchronous group communication in WANs. We designed Moshe with a fresh approach: in contrast to previously suggested WAN-oriented group membership services, Moshe does not evolve from LAN-oriented membership algorithms. Rather, it is designed explicitly for WAN environments.

We designed Moshe to address the challenges listed above. Moshe has three important novel features, each reflecting a design principle:

(1) Moshe avoids the delivery of *obsolete* views, which are views that reflect a membership that is already known to be out of date. Doing so reduces the network load during unstable periods. Furthermore, since installing a view can generate significant application overhead, avoiding the installation of obsolete views can also reduce the load generated by the application.
(2) Moshe is optimized for the common case of the underlying failure detector being relatively consistent, running a single communication round in this case.
(3) Moshe is built with a client-server design in which the membership is not maintained by every process, but only by dedicated membership servers. Such an architecture makes Moshe scalable and allows Moshe to avoid flooding the network by propagating membership updates only to where they are needed.

Each principle stands on its own and can be applied to other distributed services. The three features are further explained in Section 2.

Group membership services respond to network events (for example, 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. Moshe is built to be portable across different event notification mechanisms; the algorithm is presented in terms of an abstract notification service. The interface between Moshe and the notification service is simple, and the requirements from the notification service are very weak (see Section 3). Therefore, Moshe should be easy to build on top of most such services, including failure detectors that have each process explicitly time-out on every other process or gossip-based failure detectors [van Renesse et al. 1998].

Separating the membership service from the network event notification service greatly simplifies the design of Moshe. Furthermore, this separation allows one to configure the notification service in different ways without modifying the Moshe algorithm. Other WAN membership algorithms that do not have such a separation, like Totem [Agarwal et al. 1998] and Spread [Amir and Stanton 1998], are significantly more complex than Moshe. Such separation does exist, however, in several membership algorithms designed for LANs, for example, [Babaoğlu et al. 2001; van Renesse et al. 1994; Dolev et al. 1994; Hayden and van Renesse 1996; Malloth et al. 1995; Mishra et al. 1993; Hiltunen and Schlichting 1998].

In order to evaluate Moshe's performance over a WAN, we have implemented Moshe using CONGRESS [Anker et al. 1997], which is a distributed network event notification service suited for WANs. CONGRESS servers use an overlay network to propagate information about network events among them. CONGRESS' overlay network can be configured in different ways, and can be tuned to work better for a given WAN topology. Indeed, we were able to boost system performance by tuning CONGRESS to better suit the topology of our experiment, as shown in Section 7.

We have ran Moshe a over the Internet. Our experimental setup spanned five locations: The Hebrew University of Jerusalem, Israel; National Taiwan University; University of California, San Diego; MIT; and Cornell University. We periodically invoked the algorithm by having processes request to join or leave groups. The experiment results validate the benefits of Moshe's design principles. Specifically, we observe that Moshe terminates within one communication round in an overwhelming majority of the runs. During unstable periods, Moshe does not generate excessive traffic, and it terminates quickly after the failures are mended; we observe such unstable periods to be rare. Furthermore, we illustrate how configuring the underlying notification service to work better for a given WAN topology can boost Moshe's performance. The experiment also yields general observations regarding the performance of membership algorithms over the Internet.

Moshe is implemented as part of a novel group membership service for *computer supported cooperative work (CSCW)* applications in WANs [Anker et al. 1998]. Moshe is complemented by a virtually synchronous communication service [Keidar and Khazan 2000], and it is *partitionable* [Dolev et al. 1994; Chockler et al. 2001; Babaoğlu et al. 2001], that is, several disjoint views can exist concurrently.

Our specification of a membership service for use in a WAN preceded the design of the Moshe algorithm. In this paper, we show that Moshe implements this specification. Moshe is quite a subtle algorithm, and therefore, proving its correctness was important. In fact, in the process of proving Moshe's correctness we uncovered a case in which Moshe could deadlock; we subsequently handled this case.

The rest of this paper is organized as follows: In Section 2 we discuss the key features of Moshe. In Section 3 we describe the environment and computation model. In Section 4 we specify the guarantees of Moshe. In Section 5 we give an overview of Moshe, and in Section 6 we describe it using pseudo-code. In Section 7 we present observations and measurements from our experiments. In Section 8 we briefly describe how clients can implement virtual synchrony in conjunction with Moshe. Section 9 contains comparison with related work, and Section 10 concludes our paper. The appendix contains a proof that Moshe satisfies its specification.

2. FEATURES

The three new key features of Moshe are discussed here.

2.1 Avoiding delivery of obsolete views

Previous membership service specifications (e.g., [Dolev et al. 1994; Friedman and van Renesse 1995; Babaoğlu et al. 2001]) had included a termination property, that is, they required that every instance of the membership algorithm terminate even if the network is unstable forever. Previous membership algorithms (e.g., [Agarwal et al. 1998; Friedman and van Renesse 1995; Schiper and Ricciardi 1993; Babaoğlu et al. 2001]) satisfy the termination property, and therefore terminate even in unstable situations.

In contrast, Moshe does not deliver to an application a view that it knows to be obsolete. This means that Moshe may be non-terminating as long as the network situation remains non-transitive or constantly changes. An unstable network forces a membership service to either continuously deliver new views or else deliver none; we believe that in such situations it is better not to deliver any view. Doing so avoids network congestion due to extra view change notifications. When the network does stabilize, Moshe terminates and does not initiate new membership changes unless new network events occur. We make this property formal in Section 4.

When running Moshe over the Internet, we have occasionally observed instability periods lasting several minutes. During a two-week long experiment, we once observed a non-transitive situation that lasted half an hour (see Section 7). During this period, no changes in network connectivity occurred, and Moshe generated no messages at all. In contrast, previously suggested membership algorithms would behave as follows in this situation: They would terminate quickly (usually within seconds), delivering a view that does not correctly reflect the network situation. Shortly thereafter, they would detect the fact that the view does not reflect the network situation (e.g., by receiving an "I-am-alive" message from a process not in the view), and would then re-run the algorithm. This would be repeated over and over again for the entire non-transitive period.

It is possible to overcome lack of transitivity using relays and dynamic routing, as done, for example, in Phoenix [Malloth et al. 1995], or using a dynamic relay service like RON [Andersen et al. 2001]. Relaying can greatly reduce the risk of non-transitivity, but it cannot eliminate it entirely, as dynamic routing also takes some time to adapt (we observe this phenomenon in the experiments presented in Section 7).

In addition to the cost of running the membership algorithm multiple times, obsolete views cause extra overhead for applications that rely on virtual synchrony. For such applications, a view change may lead to sending of special messages to re-synchronize shared state (e.g., the applications in [Keidar and Dolev 1996; Fekete et al. 2001; Sussman and Marzullo 1998; Khazan et al. 1998; Amir et al. 1994; Friedman and Vaysburg 1997]). Such additional communication is especially costly in WANs. Primary-backup applications also suffer expensive penalties from view changes — a view change can initiate a lengthy recovery process in order to fail-over to a new primary. In addition, messages sent in an obsolete view will in general not be delivered by all members of the view. A message is said to be *stable* or *safe* at a group member when that member knows the message has been delivered by all view members. Many applications (examples include [Keidar and Dolev 1996; Fekete et al. 2001; Amir et al. 1994; Khazan et al. 1998]) wait for messages to become stable before they act upon them. Thus, delivering obsolete views increases network congestion by withholding information from applications that might allow them to otherwise avoid sending messages that will be discarded.

Moshe provides its applications with information about changes in network connectivity and group membership, even at times when network instability causes Moshe not to deliver a view. This information is conveyed using `startChange` events, as described in Section 4.

One consequence of Moshe is that at unstable times, there can be long periods during which the application is aware that a membership change is occurring. Typically, virtually synchronous communication services require applications to block during such periods [Friedman and van Renesse 1995]. However, there are variants of virtual synchrony that do not require such blocking, namely Weak Virtual Synchrony [Friedman and van Renesse 1995] and Optimistic Virtual Synchrony [Sussman et al. 2000]. Avoiding obsolete views is especially beneficial if processes are allowed to send messages while a view change is under way. Unlike the messages sent in obsolete views, these messages can become stable since they are not delivered until a "non-obsolete" view is delivered. Although Moshe may be useful in conjunction with any variant of virtual synchrony, we have designed it with Optimistic Virtual Synchrony in mind.

2.2 Low Message Overhead

Since message latency in WANs can 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 multicasts a single message to the other servers, and the algorithm terminates. Thus, if the maximum message latency in the network is δ, then Moshe usually terminates within δ time after all of the servers detect the change in connectivity.

However, 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 it may be necessary to run a re-synchronization round among the servers. In this case, Moshe can be delayed either by additional δ time or by additional 2δ time. Thus, in the worst case, Moshe terminates within 3δ time once network stabilization occurs and all of the servers correctly detect the network connectivity.

Typical group membership algorithm instead terminate in *all* runs 2δ time after network stabilization occurs and all of the servers correctly detect the network connectivity (for example, [Dolev et al. 1994; Agarwal et al. 1998; Ricciardi and Birman 1991; Babaoğlu et al. 2001; Hiltunen and Schlichting 1998; Schiper and Ricciardi 1993; Malloth et al. 1995]). As discussed in Section 7, our algorithm terminated in one round in almost 99% of the cases, and seldom exceeded 2δ.

2.3 A client-server design

Moshe is part of a novel architecture for group membership services designed for CSCW applications in WANs [Anker et al. 1998]. This architecture employs a client-server approach: group membership services are provided by dedicated membership servers, which themselves are not members of any multicast group. The membership servers are concerned solely with membership maintenance, and not with message transmission among group members in the different multicast groups. The processes who wish to participate as members in multicast groups act as *clients* of the membership servers. Each client is served by exactly one server at a given time; preferably, a server that is proximate to it (in the same LAN). A client sends to its server requests to *join* or *leave* particular multicast groups (these requests are handled by the notification service part of the membership server), and the membership server sends membership views to its clients. This architecture allows a Moshe server to be scalable in the number of groups and in the number of members in a group.

The membership service interface provides the hooks for clients to efficiently implement virtually synchronous communication semantics, but it does not impose such semantics. Thus, Moshe does not delay delivery of views to clients until such semantics are achieved. Clients can enforce virtual synchrony by exchanging synchronization messages among themselves; this can be done *in parallel* with Moshe's agreement on the membership view (see Section 8).

3. THE ENVIRONMENT MODEL

Moshe 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.

Moshe 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 Network event notification service

Clients use the notification service to request to join or leave groups. The notification service accumulates and disseminates failure detection information along with information about these requests. The services are provided to clients by an interface that consists of the following basic functions:

join(G) is a request to make the client a member of group G.
leave(G) is a request to remove the member from the membership of G.

Each membership server has a local notification service component that reports the client status to the membership servers 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.

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 as follows:

—the `NSView` is initially empty;

—every time a `NE` arrives, the `NSView` is set to $\texttt{NSView} \cup \texttt{NE.joining} \setminus \texttt{NE.leaving}$.

Note that the `NSView` is not a membership view, since it has no unique identifier that can be agreed upon. The `NSView` is simply the list of group members that the server currently does not suspect.

As a failure detector in an asynchronous environment, the notification service is bound to be unreliable in some runs [Chandra and Toueg 1996]: it may be inaccurate in that it may suspect correct processes. Since we wish to specify a service that can be implemented in an asynchronous environment, we do not require that the notification service be accurate. However, we assume that the notification service is always *complete*, in the sense that if a process fails to receive a message sent to it, then the process is eventually suspected. This is made formal in the **Reliable Links** property below. The liveness of Moshe depends on the notification service providing eventually consistent sets. We discuss this further in Section 4.1.

3.2 Communication guarantees

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

> 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:

> **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 clients of $S2$, or $S2$ has failed.

4. MEMBERSHIP ALGORITHM GUARANTEES

We now describe the interface between Moshe and its clients, and the service guarantees that it provides. The primary function of Moshe 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 client-server interaction is summarized in Figure 1, which also includes the interface between the clients and the notification service.

The server sends two types of events to its clients:

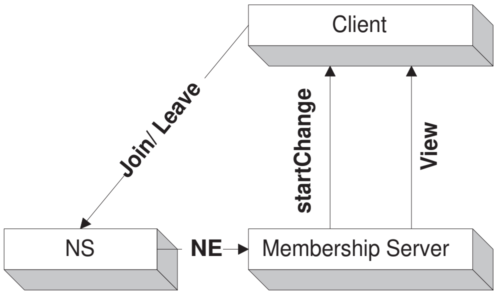


Fig. 1. The membership service client-server interface.

`startChange`(G, `startChangeNum, suggestedMemb`) indicates to the client that the server is now engaging in a membership change for group G. The view is expected to consist of the members listed in the set `suggestedMemb`.

`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.

The interface is illustrated by the following example. Two members, A and B, of group a G become connected to each other. Moshe first delivers `startChange`(G, 7, $\{A, B\}$) to A and `startChange`(G, 13, $\{A, B\}$) to B. Then Moshe delivers to both A and B `view`$(G, \langle 14, \{A, B\}, \{A \to 7, B \to 13\}\rangle)$. The `startChangeNums` mapping in the view maps A to the latest `startChangeNum` it received before the view, namely 7. Likewise, it maps B to 13.

The `startChange` event and the `startChangeNums` value of a view `V` are used in the implementation of virtual synchrony, as described in Section 8.

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`. Moshe 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 $V2$.`id` $> V1$.`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 [Chockler et al. 2001; Chandra et al. 1996]. 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 (failure detector) consistently reflects the network situation.

We first formally define what it means for the notification service to stabilize for a set of members S in a group G, and then specify property Agreement on Views, which requires Moshe to deliver the same correct view to all the members of such a stable set.

DEFINITION 4.1. *We say that* the notification service stabilizes for a set of members S in a group G *if there is a time* t_0 *such that from time* t_0 *onwards, the* `NSView` *of* G *at all of the servers serving clients in* S *is exactly* S.

> **Agreement on Views** If the notification service stabilizes for a set of members S in a group G, then eventually, all of the clients in S receive the *same* `view` V from their servers in group G such that V.`members` $= S$, and do not receive new `view` or `startChange` messages in group G henceforward.

Note that Agreement on Views defines a *partitionable* membership service: in case of partitions, the notification service can stabilize for two disjoint sets in the same group. For example, there can be two disjoint sets S_1 and S_2, so that no Moshe server serves clients both in S_1 and S_2, and from some point onward, the `NSView` of G at all of the servers serving clients in S_1 is S_1, and for servers serving clients in S_2, it is S_2. In this case, Agreement on Views requires that a view with membership S_1 be delivered to the processes in S_1, and a view with membership S_2 be delivered to the processes in S_2.

Let us now look more closely into the specification of property Agreement on Views. The 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.

(2) The notification service behaves like an *eventually perfect* failure detector (see [Chockler et al. 2001; Babaoğlu et al. 2001]), that is, it eventually stops making mistakes. A similar guarantee is formally defined in terms of network stability and failure detector properties in [Anker et al. 1998; Chockler et al. 2001].

For the sake of simplicity, in specifying property Agreement on Views, 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.

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 define 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 [Dwork et al. 1988; Guerraoui and Schiper 1997a]. This time period depends on external conditions: message latency, process scheduling and processing time. In practice (as shown in our empirical studies) stability need not last long.

[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.

4.2 Client Interface Guarantees

The `startChange` messages and `startChangeNums` are used by the clients for implementing virtual synchrony. As discussed in Section 8, to be useful they have to satisfy the following two properties:

> **Monotonicity of startChange Identifiers** The `startChange` identifiers received by each client are monotonically increasing.
>
> **Integrity of startChange Identifiers** Each `view` message V sent to a client c by a 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`, and V.`members`$=$ SM.`suggestedMembers`.

Note that a `view` message may be preceded by multiple `startChange` messages. The `members` and `startChangeNums[s]` of the view match the `suggestedMembers` and `startChangeNum` of the latest `startChange` sent before the `view`.

Note that the above properties correspond to messages sent out by the Moshe service. If the links from Moshe servers to their clients are reliable, then the same properties are viewed by the clients at their side of the link.

5. THE MEMBERSHIP ALGORITHM OVERVIEW

In this section we give an overview of Moshe. We begin in Section 5.1 by presenting the typical one-round flow. In Section 5.2 we illustrate cases in which the one-round algorithm can fail to terminate. We refer to such failure to terminate as *blocking*. The examples in Section 5.2 provide the intuition as to what mechanisms we needed to implement in Moshe in order to detect and overcome such blocking.

Moshe is 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 slow agreement algorithm is run if and only if the fast agreement algorithm is blocked. The complete algorithm is presented in pseudo-code in the next section. In the Appendix, we prove that the blocking detection mechanism detects all the cases in which the fast agreement algorithm blocks, and that the slow agreement algorithm always terminates in such cases.

5.1 The typical one-round flow of the membership algorithm

Moshe is invoked whenever it receives a `NE`. The typical message flow 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. The proposal contains the sender's `NSView`, which is the proposed membership for the next view. It also contains a `startChangeNum`, which is used by the servers to agree on the unique identifier of the view to be delivered in a manner consistent with the **View Identifier Local Monotonicity** property.

The server then waits to receive proposals with the same `NSView` from each of the servers. When all of these messages are received, the server computes the new view identifier and sends a `view` message to its clients; the membership of the new view is the `NSView` included in the proposals. Once a proposal is used for forming a `view`, it is discarded. An example of this message flow, resulting from a client B joining group of which client A is the sole member, is illustrated in Figure 2. The example proceeds in the following steps:

(1) Client B issues a join message to its local notification server (`NS`).
(2) B's notification server communicates with other notification servers via means external to Moshe (shown as a dashed line in Figure 2). This leads to a `NE` being generated at both A and B's Moshe servers.
(3) The Moshe servers send `startChange` notifications to A and B.
(4) The Moshe servers send each other proposals.
(5) Upon getting each other's proposals, the Moshe servers deliver the new `view` to A and B.

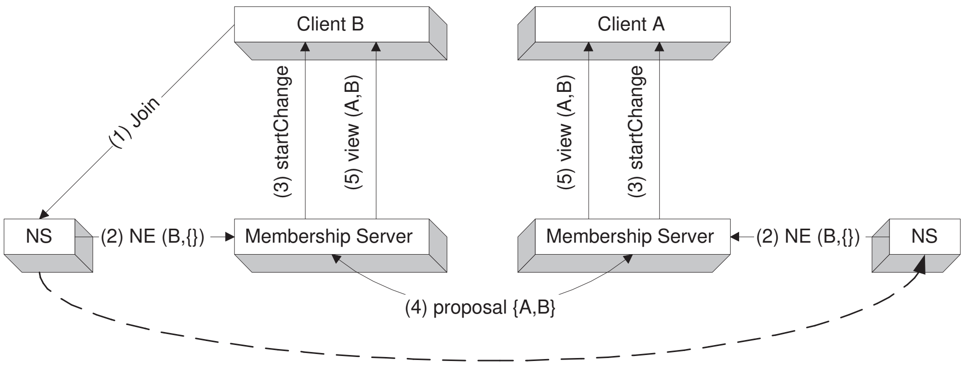


Fig. 2. The membership service typical message flow.

A one round algorithm such as this may reach agreement in a failure-free case, but cannot successfully reach agreement under all conditions. Below, we illustrate some cases in which such a one-round algorithm would lead to blocking.

5.2 Example scenarios in which the one-round algorithm would block

Example 5.1. *Initially, servers $s1$, $s2$, and $s3$ are connected. Then, due to transient congestion in the link between $s1$ and $s3$, $s1$ and $s3$ suspect each other (i.e., they receive* `NE`*s suspecting each other's clients). When the congestion passes, the suspicion is refuted and $s1$ and $s3$ both send proposals to each other and to $s2$. However, since $s2$ did not receive a* `NE`*, it does not send a proposal. In this case, $s1$ and $s3$ have begun the algorithm and sent* `startChange` *messages to their clients, but $s2$ is not running the algorithm. Thus, $s1$ and $s3$ will block waiting for a* `proposal` *from $s2$ that will never be sent, and the algorithm will never terminate, violating the* **Agreement on Views** *property.*

In this example, the blocking may be detected by some server receiving an unexpected proposal message when it did not receive a `NE`. Indeed, we detect blocking of the algorithm in such a manner. However, not all blocking cases can be detected in this simple manner, as illustrated in Example 5.2 and Figure 3.

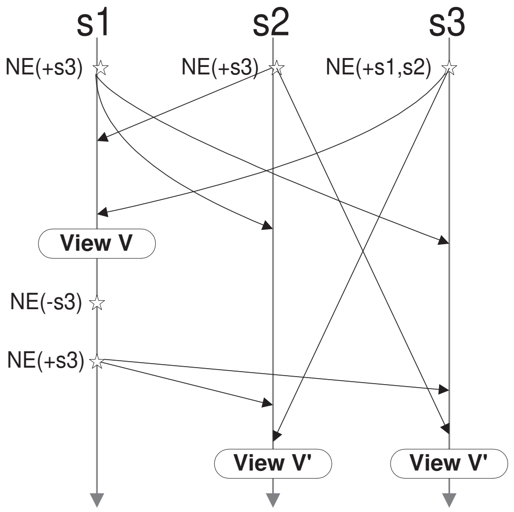


Fig. 3. A case of blocking that cannot be detected by arrival of extra messages.

EXAMPLE 5.2. *Consider three servers s1, s2, and s3. Assume that initially s1 and s2 (and their clients) are in one network component while s3 (and its clients) are in another. The two network components merge, so that s1 and s2 are both notified of the connection with s3, and s3 is notified of the connection with s1 and s2. s1 completes the one round algorithm forming a view V before s2 and s3, which are slow at receiving each other's messages. In the meantime, s1 suspects s3, but this suspicion is refuted quickly. s1 re-invokes the membership algorithm and sends proposals to the servers. Let these new proposals from s1 reach s2 before s3's original proposal and reach s3 before s2's original proposal.*

Once s2 and s3 receive each other's proposals they use the latest proposal of s1 to form a new view, V', which is different than V, and do not detect the need to start a new round. Meanwhile, s1 is blocked waiting for new proposals from s2 and s3, violating the **Agreement on Views** *property.*

In Section 6.2 below we explain how the detection mechanism detects such cases of blocking. When blocking is detected, the slow agreement algorithm is invoked.

6. THE MEMBERSHIP ALGORITHM PSEUDO-CODE

The Moshe algorithm is symmetric in that all of the servers run the same code. Therefore, we present the algorithm running at a single server. When there are changes spanning multiple groups, the same algorithm is run independently for each group. Therefore, for simplicity, we present the membership algorithm for a single group and omit the group name.

Moshe is composed of a fast agreement algorithm, a blocking detection mechanism, and a slow agreement algorithm. Both the fast and slow agreement algorithms exchange proposals, tagged with the type `FA` or `SA`, respectively. 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 them.

When the server receives a `proposal` message that 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` while running either algorithm, it begins the fast agreement algorithm anew to avoid sending an obsolete view to the clients.

The combined algorithm can be represented as a state machine with three states: a state FA in which the server is running the fast agreement algorithm, a state SA in which the server is running the slow agreement algorithm, and a state None in which the server is running neither algorithm. This state machine is shown in Figure 4.

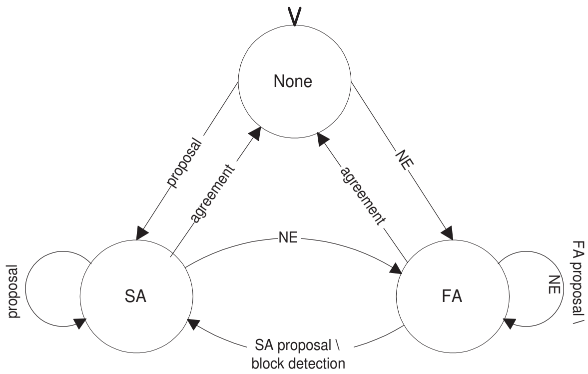


Fig. 4. The membership algorithm state diagram.

We now present Moshe in pseudo-code in three steps. We begin in Section 6.1 by presenting the fast agreement algorithm. In Section 6.2 we describe the mechanism for detecting when this algorithm blocks, and in Section 6.3 we describe the slow agreement algorithm.

6.1 The Fast Agreement Algorithm

Variables and types. Moshe uses three message types: servers send each other `proposal` messages, and send clients `startChange` and `view` messages. The types and variables used by Moshe are shown in Figure 5. 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: its value can be `FA` for the fast agreement algorithm, `SA` for slow agreement, or `none` if no algorithm is being run. `NSView` contains the aggregation of the `NEs`

```
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, CSet suggestedMembers⟩
   S→S proposal ≜ ⟨ServerId 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
   CSet         NSView = { }                          // aggregation of all NEs
   function     props [servers ↦ proposal] = null     // server' last proposal
   view         curView = ⟨0, { }, [* ↦ 0]⟩           // last view delivered
   int          startChangeNum = 0
   int          propNum = 0
   function     usedProps [servers ↦ int] = 0         // proposals used for view

Assumed external functions:
   serversOf[clients ↦  servers]
   local[CSet ↦ local clients]                        // returns local clients
```

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

Fig. 5. Types and variables for the membership algorithm.

received from the notification service. The buffer `props` is used to store the most recent `proposal` message received from every server. `curView` contains the most recent view sent to the clients. The variable `startChangeNum` ensures that the `startChange` messages sent to a client have monotonically increasing identifiers. `propNum` is a logical timestamp used to ensure that every `proposal` message sent by a server has a unique monotonically increasing identifier, and `usedProps` is used to detect if the fast agreement algorithm is blocked, as described in Section 6.2 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.

Note that the algorithm 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 per force removed from their groups. To continue participating, the clients can connect to a new server and re-join all of the groups of which they were previously members. A simple library routine can make such a fail-over transparent. Although a client that loses connection with its server cannot know whether the server has crashed or not, we assume that the server eventually also loses the connection with the client in this case[2], so having the client connect to another server is safe.

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

   if ( local(NSView) ≠ {} ) then // We only consider groups with 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

   if ( inProp.members = NSView ) then    // Proposal matches the NSView
      if ( TestIfSAProposalNeeded(inProp) ) then
         SendSAProposal(inProp)
      endif
      if ( TestIfAgreementReached() ) then
         curView = ⟨max( props[serversOf(NSView)].startChangeNum ) + 1,
                    NSView, props[serversOf(NSView)].startChangeNum⟩
         for all s ∈ serversOf(NSView)
            usedProps[s] = props[s].propNum
            props[s] = null
         end for all
         running = none
         send curView to local(NSView)
      endif
   endif

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

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

Fig. 6. 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*, that is, an event handler cannot be preempted once 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 6. Code shown in gray is not part of the fast agreement algorithm.

[2]In our implementation of Moshe we use TCP, which has this symmetric failure detection property.

The fast agreement algorithm follows the message flow described in Figure 2 above. Upon receiving a `NE`, 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:

(1) `sender` is the server that sent the `proposal` message;

(2) `members` indicates the `NSView` that this message is proposing for the new view;

(3) `startChangeNum` is used to compute the identifier of the new view.

To satisfy the **View Identifier Local Monotonicity** property, 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. 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 `proposal` messages sent by all of the servers of clients in the `NSView`. The `props` buffer collects 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, 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.

Let s be a server. Once s receives a `proposal` message proposing its own `NSView` from every server s' that has clients in the `NSView`, s sends a new `view` V to its clients. For every such server s' that has clients in the new view V, we say that s *uses* the proposal `props[s']` for V, or that the proposal `props[s']` is *used* for V. Once s sends view V to its clients, for each server s' of a client in $V.members$, s sets `props[s]` to `null` in order to avoid using the same `proposal` in future invocations of the membership algorithm. Thus, a particular proposal cannot be used for more than one view.

6.2 The detection mechanism

We now present a mechanism for detecting cases in which the fast agreement algorithm blocks. Note that we are only interested in detecting non-termination of the fast agreement algorithm in case the notification service eventually does 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 that the notification service stabilizes for the set CS, that is, that there exists a time t_0 after which the `NSView` of every server in SS is and remains CS (see Definition 4.1). 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.

We denote by $last_s$ the last `proposal` message of type `FA` sent by a server s. Since the final `NSView` of every server in SS is CS, $last_s$.`members`= CS. By the **Reliable Links** property, for every pair of servers $s, s' \in SS$, s' receives $last_s$. If every server $s' \in SS$ uses all the $last_s$ proposals (from every $s \in SS$) for the same view, 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 block is if there is some pair of servers $s, s' \in SS$, such that s does not use $last_s$ and $last_{s'}$ for the same view.

Let s be a server that does not use $last_s$ and $last_{s'}$ for the same view. As explained above, s receives $last_{s'}$. Moreover, by the `TestIfAgreementReached` condition, if s uses $last_s$ for a view, it uses some proposal from s' for the same view. Thus, there are only two possible cases:

(1) s uses $last_s$ before it receives $last_{s'}$. In this case, s uses some earlier `proposal` message from s' for the same view as $last_s$. This case occurs in Example 5.1 above where $last_{s2}$ is used (by all of the servers, and in particular, $s2$) along with proposals that were sent by $s1$ and $s3$ earlier than $last_{s1}$ and $last_{s3}$, respectively.

(2) s uses $last_{s'}$ before it sends $last_s$. In this case, s uses $last_{s'}$ for a view with an earlier `proposal` message of its own. This case occurs in Example 5.2 above where $s1$ uses $last_{s2}$ for a view, along with an earlier proposal of its own.

We now explain how our detection mechanism detects both of these cases.

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 6. The `TestIfSAProposalNeeded` function is presented in Figure 7 below. This function detects blocking if a proposal arrives when `running = none`. It also detects blocking if for the incoming proposal the entry of `usedProps` corresponding to the local server is the same as the current value of `propNum`; the code for maintaining `usedProps` is shown in gray in Figure 6.

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 $last_{s'}$ arrives, the `running` variable at s is `none`, and s detects the blocking. This case occurs in Example 5.1, where $s2$ receives $last_{s1}$ and $last_{s3}$ while $s2$ is not running the algorithm.

(2) Case 2 is detected by s' using the `usedProps` in $last_s$. If s uses $last_{s'}$ for a view before sending $last_s$, then $last_s$.`usedProps[s']` is equal to $last_{s'}$.`propNum`. Thus, when s' receives $last_s$, the condition `inProp.usedProps[me] = propNum` is true (see Figure 7), and it detects the block.
Consider Example 5.2 above. When $s1$ sends $last_{s1}$, $s1$ had already used $last_{s2}$ for a view. Therefore, $last_{s1}$.`usedProps[s2]` is equal to $last_{s2}$.`propNum`, and $s2$ detects the blocking upon receipt of $last_{s1}$.

We give a proof in the Appendix that whenever the fast agreement algorithm blocks, it is detected by the detection mechanism at some server. Furthermore, in Lemma A.5 we prove that the detection mechanism only detects blocking when the fast agreement does indeed block.

6.3 The slow agreement algorithm

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.

```
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

TestIfAgreementReached()
   if ( running = FA ) then                  // FA: all FA proposals received
      return ( ∀s ∈ serversOf(NSView) : props[s].members = NSView ∧
                                        props[s].type = FA )
   else                                      // SA: 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.

Fig. 7. Function definitions for the membership algorithm.

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 `proposal` message (of any type) this server has previously 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 it by also sending a `proposal` message of type `SA` and a `propNum` value *equal to* the maximal value of `propNum` in any `proposal` message this server previously sent or received. Ideally, this value will be equal to the `propNum` in the `proposal` from the initiating process (hereafter the *initiator*)[3], and all the servers' `proposal` messages will have 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 sending a new `SA proposal` with the value, and storing this value in its local `propNum`.

Note that there may be several initiators. The difference between initiating a round of the slow agreement algorithm and joining a round is that servers joining a round do not increase the `propNum` to be larger than the highest value they received. Thus, once all the servers are running the slow agreement algorithm and no further `NE`s occur, 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 such `proposal` messages are collected from all of the servers, the slow agreement algorithm terminates.

In Figure 7, we complete the pseudo-code shown in Figure 6 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 (see Figure 6). The function `SendSAProposal` is also used by the slow agreement algorithm to join a round in progress.

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 `TestIfAgreementReached`. In Figure 7 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.

7. EXPERIMENTAL RESULTS OVER THE INTERNET

In the previous sections we have presented the Moshe algorithm; the Moshe algorithm uses an *abstract* notification service and a reliable FIFO communications layer. An implementation of Moshe has to instantiate the abstract notification service with an actual one, and has to employ some form of reliable communication. We implemented Moshe using the CONGRESS WAN-oriented notification service [Anker et al. 1997], so that CONGRESS served both as the notification service and as the reliable

[3]If the initiator receives the last `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.

FIFO communications layer for Moshe.

CONGRESS internally implements a reliable communications service, which it uses to detect communications failures and to propagate network events. The communications service is built using an overlay of TCP/IP streams between CONGRESS servers, with the exact interconnection topology determined by an initial configuration description. A CONGRESS server suspects its neighboring server when the TCP link between them goes down. Since TCP is fine-tuned to have few false suspicions, CONGRESS provides a similar quality of service. When implementing Moshe on top of CONGRESS, we exploited the CONGRESS communications service to reliably send messages among Moshe servers: Moshe servers communicate among themselves using the sockets used by the CONGRESS servers, so that CONGRESS and Moshe messages are sent on the same links.

In this section we present performance measurements from running the system consisting of both CONGRESS and Moshe over the Internet.

Our goal was to evaluate, in a realistic setting, three important design decisions employed by Moshe: first, Moshe optimizes for situations in which the fast algorithm terminates successfully; second, Moshe does not deliver obsolete views; and third, Moshe benefits from being built on top of a notification service.

The first design decision, optimizing for the fast case, is justified if the number of cases in which the fast agreement algorithm is run significantly exceeds the number of cases in which the slow agreement algorithm is run. We therefore measured the number of times each of the two algorithms is run during a long-term experiment in a realistic setting. Our design decision to refrain for delivering obsolete views at unstable periods can cause Moshe to wait a long period after a notification event before delivering a view. Therefore, in order to evaluate our policy of not delivering obsolete views, we measured how often Moshe waits a long period (more than 4 seconds) after a notification event before delivering a view. To evaluate the utility of building Moshe atop a notification service, we measured how the performance of Moshe benefits from configuring the notification service to specific network conditions.

We ran the service over the Internet in five locations: MIT, UCSD, Cornell University (CU), the Hebrew University of Jerusalem, Israel (HUJI), and National Taiwan University (NTU) in Taipei, Taiwan. We ran the service for a total of almost two weeks – ten days in one configuration, and two and a half days in another. We now report on our observations during this experiment.

In Section 7.1 we study the nature of the network the experiments ran on. Then, in Section 7.2 we describe the experiment we ran. In Section 7.3, we describe the events that occurred during the experiment – machine failures, network partitions, etc. In Section 7.4, we report on the number of times the fast and slow agreement algorithms were run. In Section 7.5 we examine the frequency of cases in which it takes Moshe more than 4 seconds to deliver a view. Finally, in Section 7.6, we discuss the running time of Moshe.

7.1 The network situation

In order to understand the nature of the network we were running on, we used 'ping' to measure the round-trip times and loss rates among pairs of processes[4]. We had 'ping' send a message once every minute. We ran 'ping' for 68 hours from CU and UCSD, for 57 hours from MIT, and for 30 hours from NTU. Since HUJI is behind a firewall that does not let 'ping' messages pass through, we instead measured the loss rate and round-trip times to a gateway machine at HUJI that is not behind the firewall. We could not run 'ping' from HUJI.

From	MIT		UCSD		CU		NTU	
To	no bursts	all	no bursts	all	no bursts	all	no bursts	all
MIT	–	–	0.6%	0.7%	0.3%	0.5%	1.0%	1.3%
UCSD	1.3%	1.5%	–	–	0.5%	0.8%	1.3%	1.3%
CU	1.8%	2%	0.7%	1.0%	–	–	0.7%	0.7%
NTU	1.7%	1.8%	1.4%	1.7%	1.3%	1.9%	–	–
HUJI	1.5%	1.9%	0.7%	0.8%	0.3%	0.6%	1.4%	1.7%

Table 1. Loss rates measured by 'ping'.

On occasion, two machines would become disconnected for several minutes, leading to a sequence of three or more messages being lost. We want to distinguish such long-term disconnections from single packet losses. Therefore, in addition to computing the total percentage of messages that were lost, we also computed a *no bursts* loss rate, excluding bursts of three or more consecutive losses. That is, messages lost in bursts of three or more are not counted as lost in the no burst loss rate. In Table 1 we show the measured loss rates – both the normal count, and the no bursts loss rate. The difference between the total loss rate and the no bursts loss rate gives the percentage of messages that were lost in bursts of three or more – that is, messages lost during disconnections.

The observed loss rates varied greatly with time. For example, in Figure 8 we show the cumulative number of losses from MIT to CU observed over a 57 hour period. In this experiment, there were three bursts of three losses and no longer loss bursts. Thus, only 9 of the 73 losses were a part of a burst. During the first 19.5 hours that 'ping' was running, only one message (out of 1161) was lost. Then, during the next half hour, 10 of 34 messages were lost for a loss rate of 29%. The loss rate then plunged to zero again, for seven hours. Starting at the 27th hour of the experiment, the link became lossy again. For the following 5.5 hours, the loss rate (computed over the entire 5.5 hours) was 12.9%. This illustrates how unpredictable loss rates over the Internet can be.

We assume that the loss rate observed during the second day is unusual, and usually the link between MIT and CU is more reliable than the links to NTU and HUJI from these sites, as reflected by the first day of the measurement from MIT to CU and by the measurements from CU to MIT.

We measured the median, average, minimum, and maximum round trip times encountered by 'ping'. The results appear in Table 2. The round trip times were

[4]These measurements were not done at the same time as our experiments with Moshe.

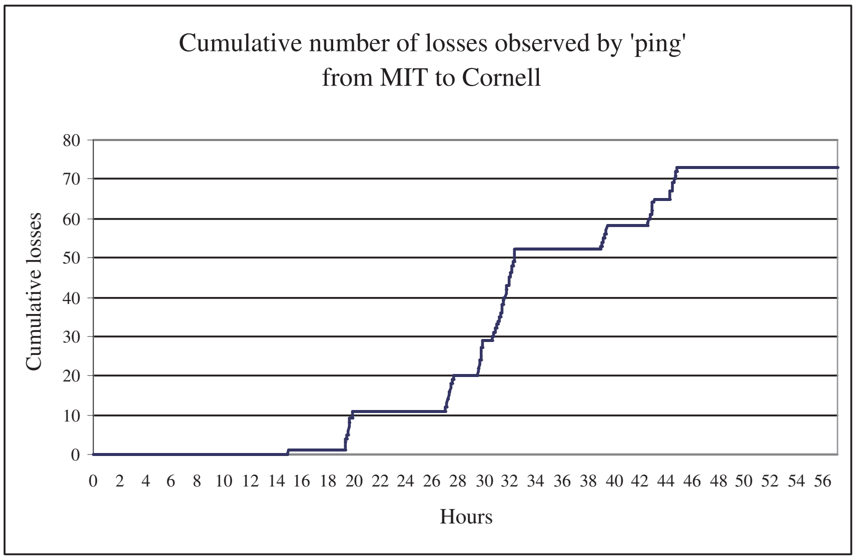


Fig. 8. Cumulative losses observed by 'ping' from MIT to Cornell.

fairly stable. Occasionally, a message takes much longer than the average, but such messages are rare. Hence, the average time is usually very close to the median. The minimum time was also usually close to the median. An exception was the time from CU to HUJI, which was usually around 582 ms., but went down to around 170 – 200 ms. for roughly one hour of the 68-hour measurement period.

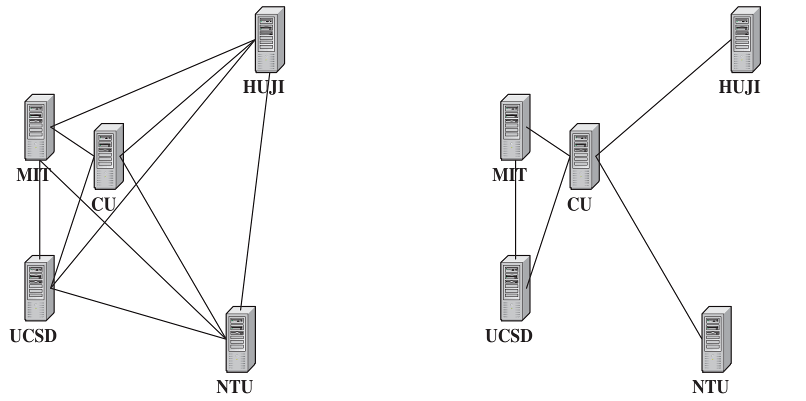
From To	MIT med/avg/min/max	UCSD med/avg/min/max	CU med/avg/min/max	NTU med/avg/min/max
MIT	—	91/98/87/2803	19/24 /16/2214	235/239/231/1327
UCSD	90/96/87/653	—	85/89/80 /1778	272/275/266/1272
CU	19/23/16/1225	85/90/80/3711	—	236/237/235/266
NTU	234/239/231/1737	272/277/228/3114	236/237/235/348	—
HUJI	584/587/580/1384	619/618/235/3471	586/582/170/609	763/758/412/842

Table 2. Round-trip times in milliseconds measured by 'ping', median, average, minimum, and maximum values.

As seen in Table 2, the measured round-trip times between MIT and the other locations were as follows: to CU, around 20 ms.; to UCSD, around 90 ms.; to NTU, around 235 ms.; and to HUJI, around 585 ms. HUJI had the longest round-trip times to all destinations. The longest measured average round-trip time was between NTU and HUJI, which was around 750 ms.

7.2 The experiment setup

At each of the five locations, we ran a membership server and a program simulating ten clients. The clients generated activity in ten groups.



Configuration 1: Fully connected. Configuration 2: Nexus at Cornell.

Fig. 9. Two CONGRESS configurations we experimented with.

When using CONGRESS, one has to configure a logical topology, such that servers communicate only with their neighboring servers in this topology. We experimented with two different configurations. First, we configured CONGRESS so that all of the servers would communicate with each other directly, that is, all would be neighbors in the logical topology. This configuration is illustrated in Figure 9(a). We ran Moshe in this configuration for 10 days. We then configured CONGRESS so that CU serves as a nexus for HUJI and NTU, as illustrated in Figure 9(b): the three servers in the US communicate with each other directly, and the servers at NTU and HUJI communicate directly only with CU.

The first configuration maximizes the chance for non-transitive communication, which may lead to the slow agreement algorithm being invoked. Moreover, lengthy non-transitivity may lead to unstable periods. With the second configuration, non-transitivity is very rare. The second configuration also eliminates the least reliable links (please see Table 1), minimizing the probability that Moshe will be delayed due to message loss.

We periodically invoked Moshe by having a client request to join or leave one of the ten groups. The pseudo-code of the client simulation program is presented in Figure 10. It has two phases, an *initialization phase* and a *running phase*. During the initialization phase ten clients are started. The time that elapses between two client beginning is at most three minutes. This simulates clients starting roughly at the same time. Each client joins each group with probability 0.2. Thus, following the initialization phase, there is an average of two members per location in each group.

During the running phase, events occur in batches of one to five at a time, with up to a half hour between batches. This ensured that the overhead would be low; we had to be friendly to the machines that hosted our experiment. Batches modeled what we believed would be the common case: users often perform a number of actions at the same time. This also allowed us to study the effect on other groups of a server handling another group's events.

Most of the time, each of the groups had members at all locations. That is, in most of the invocations of the algorithm, all the servers participated because each

```
function groups [process ↦ groups]     // maps process to groups it is in

// Initialization phase
for all c ∈ { 1 .. 10 }
   create client c
   for all g ∈ { 1 .. 10 }
      choose f ∈ { 1 .. 5 }
      if ( f = 1 )
         have c join g
      endif
   choose t ∈ { 1 .. 180 }
   sleep t seconds
end for

// Running phase
do forever:
   choose loops ∈ { 1 .. 5 }
   for all loop ∈ { 1 .. loops }
      choose act ∈ { join, leave }
      if ( act = join ∧ { 1 .. 10 } - groups[c] ≠ {} )
         choose g ∈ { 1 .. 10 } - groups[c]
         have c join g
      else if ( act = leave ∧ groups[c] ≠ {} )
         choose g ∈ groups[c]
         have c leave g
      endif
   end for
   choose t ∈ { 1 .. 1800 }
   sleep t seconds
end do
```

Fig. 10. The client simulation program pseudo-code.

of them had at least one client which was a member of the group in which the membership change was taking place. On occasion, a group did not have members at all of the locations.

7.3 The events

We now describe the network events that occurred during the experiments.

7.3.1 *Configuration 1.* During the ten days of the experiment with the first configuration, the server at MIT delivered 10,786 views to its clients. We observed several temporary communication failures. Most of the observed failures were non-transitive, for example, the link between NTU and HUJI would be down, but both NTU and HUJI could communicate with MIT. The longest transient communication failure lasted 26 minutes. On one occasion, we observed a full partition, where NTU was isolated from the other four locations for roughly an hour and a half. The partition was not detected at the same time at all of the members; the first server detected that NTU was disconnected roughly a half hour before the other servers had all detected it. On two occasions, two of the membership servers failed due to software errors or due to the crashing of the terminal from which the programs were run. They were soon restarted, along with the respective client simulation programs. In both cases, the three surviving servers ran uninterrupted.

During periods with non-transitive communication, Moshe does not generate additional traffic, and by design, does not terminate until the non-transitivity passes. During the full partition, Moshe servers at both network components continued delivering views: the server at NTU installed views with local members only, and the other four servers delivered common views without NTU members.

7.3.2 *Configuration 2.* After one day of the second experiment, the machine we ran on at CU crashed for several hours due to a hardware problem. During these hours, the NTU and HUJI servers each operated by itself, and MIT communicated only with UCSD, although nothing was wrong with the Internet connection among all four sites. The partition occurred because the overlay network used by CONGRESS is static, and we configured it to route all messages from NTU and HUJI to other sites via CU. When the machine at CU recovered, the partition merged.

This illustrates a drawback of using CONGRESS with such a configuration: it makes the system susceptible to a single point of failure. There are plans to make CONGRESS more robust by making the overlay dynamic, and thus allowing fail-over in such cases: upon detecting that the nexus is down, servers will try to connect via a surrogate nexus. Had this change been made in CONGRESS, the NTU and HUJI servers would have connected via MIT or UCSD, and the partition would have been avoided. This is a simple change to make. However, it is not in the scope of our project, which focuses on Moshe.

During the partition, invocations of Moshe involved only one or two servers, and thus were not representative. Therefore, for the sake of studying Moshe's performance below, we ignore them. Excluding views delivered during the partition, the MIT server delivered 2,559 views in the second configuration.

7.4 The number of slow agreement cases

Of the 10,786 views the MIT server delivered to its clients in the first experiment, only 125 were resolved by the slow agreement algorithm. Thus, 98.84% of the invocations were resolved using the fast agreement algorithm. The percentage of slow agreement cases was quite stable throughout the execution. The numbers at the other servers were similar, as shown in Table 3. Recall that these results were obtained when CONGRESS was configured to maximize the chance of the slow agreement algorithm being invoked. We see these results as overwhelming evidence of the benefit of our design, which optimizes for situations that can be resolved using the fast agreement algorithm.

In the second experiment, the number of invocations of the slow agreement algorithm drops by an order of magnitude: only 4 of the 2,559 views at MIT were resolved using the slow agreement algorithm, while 99.84% of the cases were resolved using the fast agreement algorithm. Similar numbers were observed at the other servers, as shown in Table 4. Recall that in this experiment, only the three US locations are connected with each other directly, and non-transitivity can occur only among these three servers. Since the slow agreement algorithm is only invoked if non-transitivity occurs, it is seldom invoked in this configuration.

Server Location	Total Number of Views	Number of Slow Algorithm Cases	% Slow Algorithm	Number of Fast Algorithm Cases	% Fast Algorithm
MIT	10,786	125	1.16%	10,661	98.84%
UCSD	9,701	116	1.20%	9,585	98.80%
CU	9,484	104	1.10%	9,380	98.90%
NTU	10,392	107	1.03%	10,285	98.97%
HUJI	8,802	101	1.15%	8,701	98.85%

Table 3. Views resolved by the fast and slow agreement algorithms, first configuration.

Server Location	Total Number of Views	Number of Slow Algorithm Cases	% Slow Algorithm	Number of Fast Algorithm Cases	% Fast Algorithm
MIT	2,559	4	0.16%	2,555	99.84%
UCSD	2,281	4	0.18%	2,277	99.82%
CU	2,338	5	0.21%	2,333	99.79%
NTU	2,642	5	0.19%	2,637	99.81%
HUJI	2,542	4	0.16%	2,538	99.84%

Table 4. Views resolved by the fast and slow agreement algorithms, second configuration.

Although the tests were run at only five locations, we believe that the results generalize to a larger number of locations. Even with many locations, one would typically configure CONGRESS with no more than five locations directly connected to each other. As illustrated by the difference between the two experiments, reducing the number of servers directly connected to each other significantly reduces the slow agreement cases, even if the number of participating servers remains constant. Therefore, we believe that with a large number of locations configured so that only a handful are directly connected, similar results would be obtained. We are unable to verify this hypothesis, however, due to lack of resources.

7.5 Long delays in view delivery

Moshe, by design, does not terminate while the network conditions are unstable. There are two situations that we classify as unstable: (1) when the network situation is constantly changing; and (2) when the failure detector outputs of connected processes differ. The latter occurs if the underlying communication is not transitive, for example if MIT is connected to NTU and UCSD, while UCSD and NTU are not connected. Non-transitivity can be overcome using relays; however, it is up to the communication layer to detect non-transitive cases and relay traffic through active links. Moshe runs atop a communication layer, and reflects its status. Thus, Moshe waits for the communication layer to establish relays that would overcome the non-transitivity. In the interim, Moshe does not deliver new views.

In order to study the length of unstable periods, we examine the *total running time* of Moshe at each server. We define the total running time as the time from the first `NE` that occurs at this server after a view until the next view is sent by this server to its clients. Note that multiple `NE`s can occur while the algorithm is in progress, before a view is actually sent to the clients. Thus, this measurement does not capture the time Moshe takes to resolve one network event; this latter time is studied in the next section. Rather, the total running time is the length of the period during which clients are aware that a view change is in progress. Given Moshe's policy not to deliver obsolete views, an unstable period of a certain length will be reflected in a total running time of similar length. We now discuss cases in which the total running time exceeded 4 seconds.

In the first configuration, only 379 of the 10,786 views at MIT, (3.5%), were delivered 4 seconds or more after the first `NE`. Only 167, (roughly 1.5%), lasted 20 seconds or longer. The median total running time was 1129 ms., which was the same as the average excluding cases over 4 seconds. The total running times over 4 seconds were very sparsely distributed. The maximum running time of 32 minutes was observed when NTU disconnected from all the servers except MIT, and continued to communicate with MIT for 32 more minutes before disconnecting from it too. The second longest total running time was 26 minutes.

In the second configuration there were fewer unstable periods: for only 14 of the 2,559 views at MIT, (0.5%), the total running time was 4 seconds or longer, and the longest total running time was 31 seconds. The median total running time was 680 ms., and the average excluding cases over 4 seconds was 814 ms. Unstable periods are less frequent in this configuration since non-transitivity is less likely.

7.6 Performance measurements

In this section we study the *duration* of Moshe executions, the time from the *last* `NE` received from CONGRESS before the view delivery until the Moshe server sends the view out to its clients. This is the time Moshe takes to resolve the last notification event before the view. We only consider executions of up to 4 seconds; we assume that longer executions pertain to unstable periods, as discussed above.

At MIT, 97% of the executions had durations up to 4 seconds. When comparing this figure with the results presented in Section 7.5 above, we see that in most cases when the total running time of Moshe exceeded 4 seconds, the duration from the last NE to the view also exceeded 4 seconds. This is typical in unstable situations where the network is slow to adapt. Consider the following example: the site at Taiwan is disconnected from the other sites for a while, and then, when it comes up, its link with MIT is re-established more than 4 seconds before its link with UCSD is re-established. MIT gets exactly one NE, when its link with Taiwan is re-established, and so its duration and total running time are both the same, and are both over 4 seconds as the view cannot be delivered before UCSD also re-establishes connectivity with Taiwan. This behavior is typical on the Internet, where routing tables do not simultaneously adapt to reflect the correct network situation.

Before we present our measurements, let us first examine what we would expect the typical duration to be, for example, at MIT. Let a join or leave event occur at some site, which we call the *origin*. The CONGRESS server at the origin sends a notification about the join to all of the other servers. Once this notification reaches MIT, a `NE` occurs. In order for Moshe to complete at MIT, the join notification has to first reach all of the servers, causing them to send proposals, and then the proposals have to be received at MIT. Thus, in the absence of message loss, the duration of Moshe should be roughly the one-way time from the origin to the most remote server plus the one-way time from the most remote server to MIT, minus the one-way time from the origin to MIT. For example, if the origin is CU, this figure would be around 560 ms. If the origin is MIT, it would be around 585 ms. Message loss can, of course, cause further delays.

In Section 7.6.1 we present measurements of Moshe's duration at MIT; these were similar to measurements collected at the other two US sites. In Section 7.6.2 we compare the duration of invocations resolved by the fast and slow agreement algorithms, also at MIT. Section 7.6.3 presents measurements collected at HUJI, and explains how and why they differ from those collected in the US.

7.6.1 *Moshe duration at MIT.* In the first configuration, the duration of Moshe was not longer than 4 seconds for 97% of the views delivered. The median duration of Moshe at MIT in this configuration was 1112 ms., and the average duration, computed for cases up to 4 seconds, was 1118 ms. In the second configuration, the duration was closer to the expected value computed above: the median duration was 670 ms., and the average, excluding the 8 cases over 4 seconds was 797 ms. We now elaborate on our observations in the two configurations.

7.6.1.1 *Moshe duration distribution – configuration 1.* A histogram of Moshe duration (values up to 4 seconds) is shown in Figure 11. Notably, the duration is distributed around diminishing peaks. The first large peak centered at around 650 ms., the second, around 1250 ms., the third, around 1800 ms., and the fourth, around 2300 ms. There is also a small peak around 250 ms., which is due to events initiated at HUJI, as explained in Section 7.6.3 below.

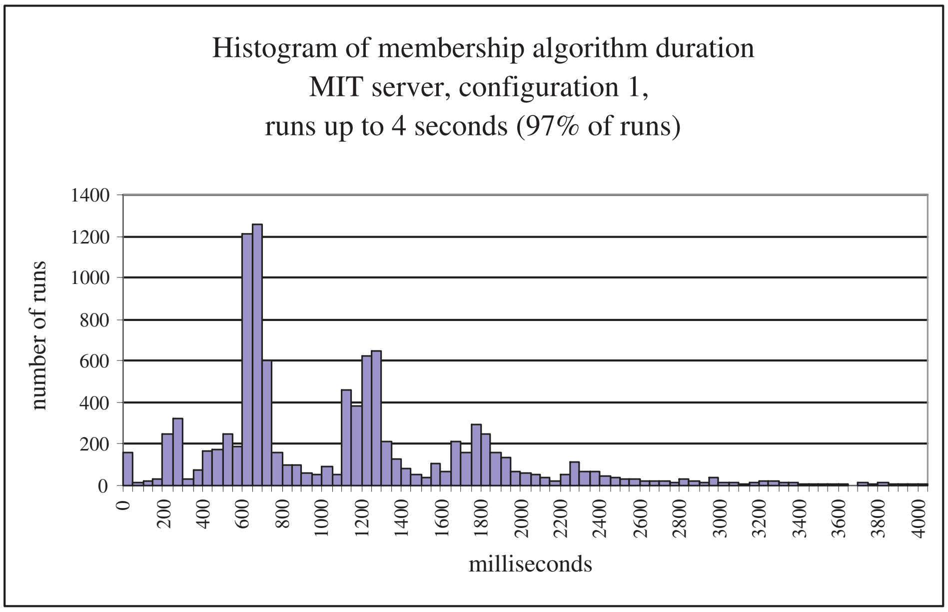


Fig. 11. A histogram of Moshe duration at MIT, first configuration.

The first and highest peak, around 650 ms. is somewhat larger than our expectation for the loss-free case, but taking into account processing and scheduling time, it is quite reasonable. The subsequent peaks are due to message loss and TCP retransmissions. Recall that in this experiment messages were sent over TCP/IP connections between each pair of servers. If a message sent over a TCP link is lost, the message is retransmitted after a timeout which is typically the estimated round-trip time on the link plus twice the standard deviation of the round-trip time. Considering the round-trip times in Table 2, 500 – 600 ms. are reasonable retransmission timeouts in our environment.

To analyze the probability of delay due to message loss, let us examine the message flow involved in an invocation of Moshe. Moshe is usually invoked when a process is joining or leaving a group. The join or leave request is issued at some server. CONGRESS uses TCP/IP links to propagate the information about the join or leave to all servers. Thus, CONGRESS sends four messages over TCP/IP links. When these messages are received, a `NE` occurs at all of the servers. Upon receiving the `NE`, each Moshe server sends `proposal` messages to the other servers, again over TCP/IP links. For Moshe to complete at the MIT server, `proposal` messages from the other four servers have to arrive at MIT. The completion of Moshe may be delayed when any of these eight messages – a CONGRESS notification or a `proposal` – is delayed due to loss.

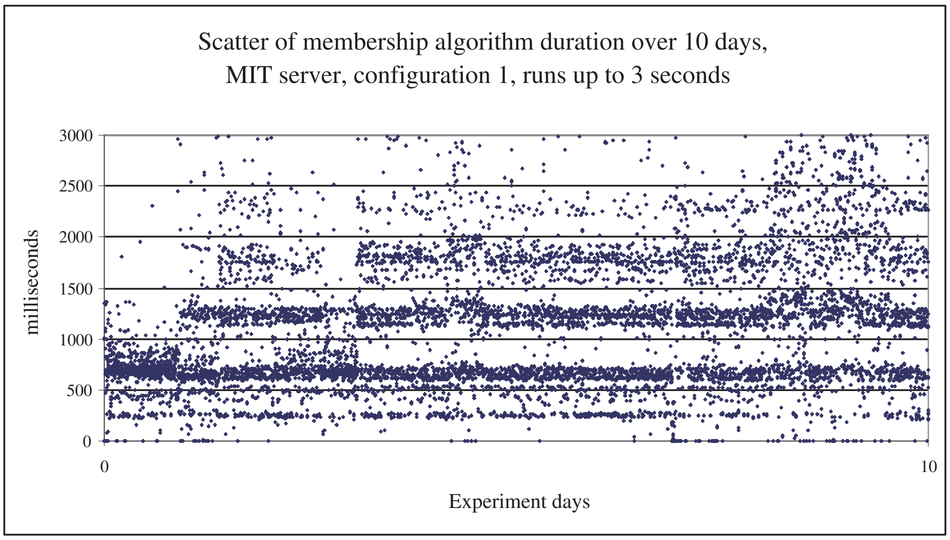


Fig. 12. Distribution of Moshe duration at MIT over time, first configuration.

As observed in Section 7.1 above, loss rates on the Internet greatly vary with time. This causes the duration of Moshe to also vary with time. In Figure 12 we show how the duration of Moshe was distributed over the 10 days of the experiment. We observe that in the last two days there was an increase of cases in which Moshe took longer to complete. We assume that this is caused by the network conditions deteriorating during these days.

Approximately 50% of the runs in the first configuration lasted over 1100 ms. In order to approximate the percentage of cases that were delayed due to message loss, we excluded runs due to *first join events* (cf. 7.6.1.4 below) and runs resolved by the slow agreement algorithm, since such runs lasted longer regardless of loss. We also excluded runs exceeding three seconds, assuming that such delays were caused due to instability, for example, lack of transitivity in the network. Of the remaining runs, roughly 46% lasted over 1100 ms. If we exclude the last two days of the experiment, during which the network was highly unstable, this number goes down to 40%.

Still, this is a very high percentage. If Moshe is delayed only due to the loss of one of eight messages, as explained above, then the 40% figure would imply that each message is lost with a probability of approximately 6.3%, assuming independence of message loss on different links. Although we have observed in Section 7.1 that loss rates vary greatly, this still seemed to be too high: it is more than double the highest loss rate we observed over two or three days by running 'ping'. We therefore hypothesized that there were more than eight messages actually being sent, that is, that messages were being broken up by TCP into more than one packet.

In order to verify this hypothesis, we tracked the packets actually being sent using 'tcpdump'. As expected, we observed that messages were usually being split into two packets. This is because the internal CONGRESS mechanism for sending messages executes two 'write' calls in order to send a single message over the TCP/IP socket: first, it writes the length of the packet (four bytes), and next, it writes the data. Since the TCP/IP links were relatively idle, the first four bytes would be sent by TCP immediately in a separate IP packet, and the rest of the message would be sent in a second packet. We believe that changing CONGRESS to execute a single 'write' would improve the performance. However, making changes to CONGRESS are outside our scope, we merely used CONGRESS to implement Moshe; we hope that such a change to CONGRESS will soon be made.

7.6.1.2 *The combined duration of Moshe and* CONGRESS. The duration of Moshe depicted in Figure 11 above is measured from the time the MIT server gets a `NE`, until the server gets the corresponding view change. Therefore, it does not capture the time it takes CONGRESS to generate a `NE` from the time an actual event, such as a process join or leave occurs.

Let us examine the time it takes CONGRESS to generate a `NE`. When a client wants to join or leave a group, it sends a message to its local server. Since clients are served by servers in their LAN, the time it takes for this message to reach the CONGRESS server is negligible (typically less than one millisecond). When the server receives such a message, it immediately generates a `NE` at the local Moshe server, and sends a multicast message through the CONGRESS overlay to the other servers. In general, the time it takes CONGRESS, or any other notification service, to generate a `NE` directly depends on the time it takes the pertinent information to propagate through the network.

Specifically, the combined duration of Moshe and CONGRESS at MIT for events originated at MIT, is practically the same as the duration of Moshe measured for these events at MIT. The histogram in Figure 13 shows the duration of Moshe in a subset of the runs of Figure 11, consisting of runs originated at MIT. We observe that it exhibits a similar pattern to the histogram of Figure 11.

7.6.1.3 *Moshe duration distribution – configuration 2.* In the second configuration, the most lossy links to HUJI and NTU are eliminated. This causes messages to and from these locations to traverse two more reliable links instead of a single less reliable link. Eliminating lossy links reduced to a half the number of times Moshe

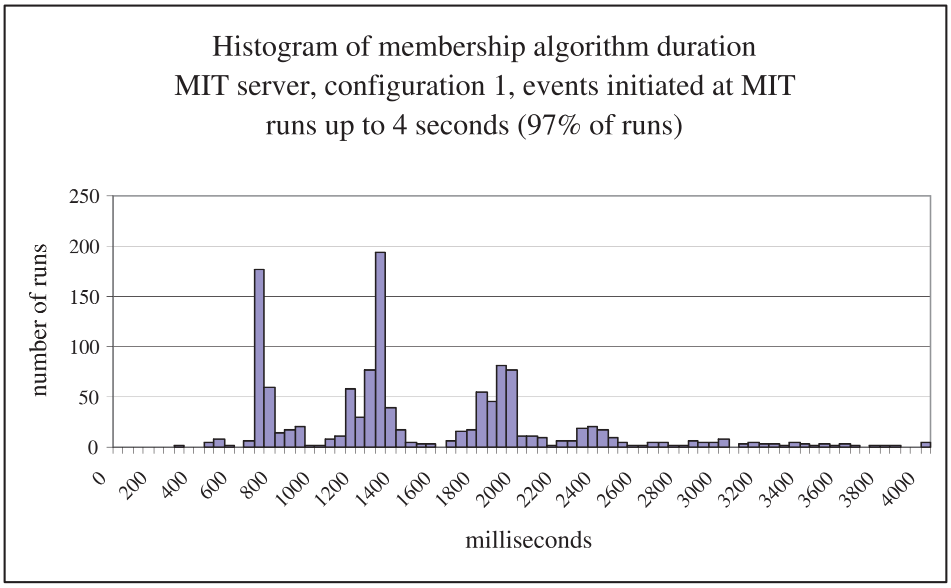


Fig. 13. A histogram of Moshe duration at MIT for events originated at MIT, first configuration.

was delayed due to loss: in this experiment, the running time of Moshe exceeded 1100 ms. for only 629 of the 2,559 views, under 25%. The histogram of the duration of Moshe at MIT during the experiment with the second configuration is shown in Figure 14. Notably, the peaks for this configuration are still centered around the same values as in the first configuration; they differ in their relative sizes.

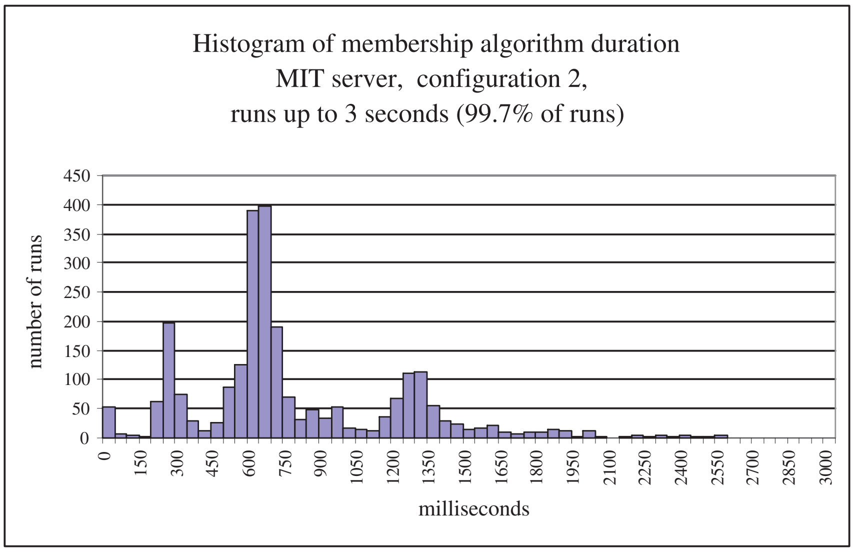


Fig. 14. A histogram of Moshe duration at MIT, second configuration.

This again illustrates the importance of better configuring the notification and communication services in order to boost Moshe's performance.

7.6.1.4 *First join events.* Moshe was invoked due to different events - join, leave, server failure, etc. We call a *first join* the case where a client joins a group for which no other client of its server is a member. The measured duration distributions for most of the event types were similar, with the exception of first join events. The duration of Moshe for first join events of local members (i.e., at MIT), was three orders of magnitude smaller than for other events – it averaged less than one millisecond. In contrast, the duration of Moshe for first join events of remote members (i.e., the joining member is not at MIT) was higher than for other events, by about 50%: In the first experiment, 242 of 10,786 runs of Moshe were due to a remote first join. For these runs, the median duration was 1765 ms., and the average excluding cases over 4 seconds was 1756 ms.

First join events are special, since in these cases, the local CONGRESS server does not have information about group membership[5]. Therefore, CONGRESS cannot locally issue a `NE` immediately upon receiving the join, but instead has to query the other CONGRESS servers to learn of the current group membership. The query is sent to the other servers together with the report about the join. When this report is received by the remote servers, it leads to a `NE`, and the servers send each other proposal messages. The proposals are transmitted roughly at the same time as the query response. Finally, when the query response arrives at the local server, a `NE` is generated locally, and a proposal is sent by the local server. At this time proposals from remote servers have already arrived and the view is ready to be immediately delivered (within less than a millisecond). The remote servers, on the other hand, cannot deliver the view until the proposal from the local server arrives.

7.6.2 *Comparing the duration of the fast and slow algorithms.* Recall that the slow agreement algorithm is invoked when it is detected that the fast agreement algorithm is blocked. We distinguish between the following two cases:

(1) the slow agreement algorithm is invoked at a site where there was a preceding `NE`. At this site, the fast agreement algorithm is first invoked, executing a message round. Then, the slow agreement algorithm is run as well, executing another round or two, depending on whether the `propNum` values of the participants are initially the same. The measured duration for this case spans both the fast and slow agreement algorithms.

(2) the slow agreement algorithm is invoked when an unexpected proposal is received while the algorithm is not locally running. In this case the fast algorithm is not run at this site at all. For this case, we measure the duration from the time it is detected that the algorithm should run (by receipt of an unexpected proposal), and until a view is sent. This spans only the slow algorithm.

In Figure 15 we show a histogram of the running times of the slow agreement algorithm at MIT, for the first configuration. We distinguish between the two cases described above. For cases in which the algorithm was preceded by a `NE`, the median

[5] CONGRESS only disseminates information about a group to servers that have members in this group.

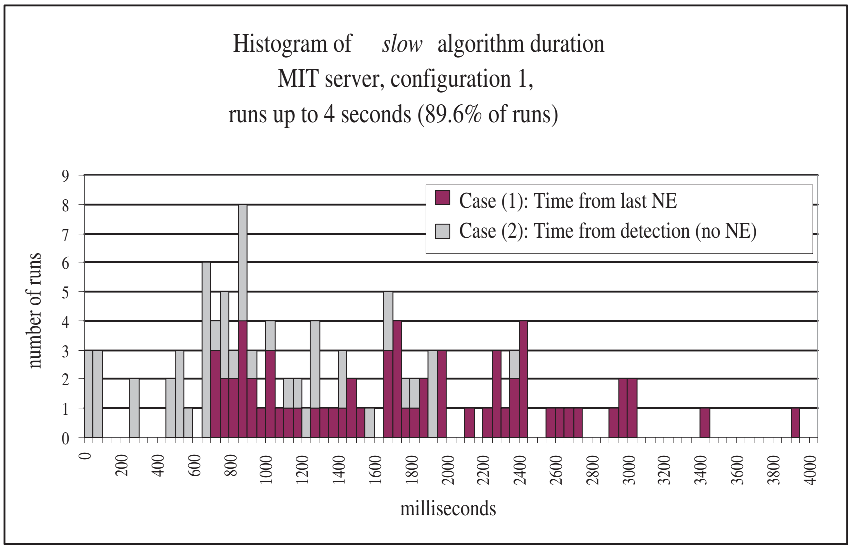


Fig. 15. A histogram of the slow agreement algorithm duration at MIT, first configuration.

duration was 1865 ms., and the average excluding cases over 4 seconds was 1776 ms. This is about 60% longer than the median and average duration of Moshe for all runs, dominantly fast agreement cases. The first peak in the distribution of these running times appears to be centered at approximately 900 ms., which is 250 ms., or 40% more than the peak for all runs. For cases in which there was no preceding `NE`, the median algorithm duration was 871 ms. and the average excluding cases over 4 seconds was 922 ms. This is about 80% of the usual duration.

Based on these numbers we hypothesize that in most cases, the slow agreement algorithm involves one message round (in addition to the one round of the fast agreement algorithm) and not two. Note that two rounds should not last twice as much as one, since the time for propagating event notifications to remote sites is also part of the running time. A two round algorithm should be longer than the one-round algorithm by roughly the one-way time to the most remote site. This is close to our observations.

7.6.3 *Moshe duration at HUJI.* The measurements gathered at MIT were typical for the US sites. At HUJI and NTU, however, the duration of Moshe was, on average, shorter. In Figures 16 and 17 we show the distribution of Moshe's duration at HUJI for the two configurations. The median duration for the first configuration was only 750 ms., and the average excluding cases over 4 seconds was 906 ms. This difference stems from the fact that HUJI is the farthest location – the round-trip times between it and other locations are the longest (please see Table 2). Therefore, membership events other than those initiated at HUJI are reported at HUJI later than at other locations. This is illustrated by the following example:

Example 7.1. *Consider Moshe being run by three servers: MIT, CU, and HUJI. Assume that messages from CU reach MIT in 10ms., and messages from both CU*

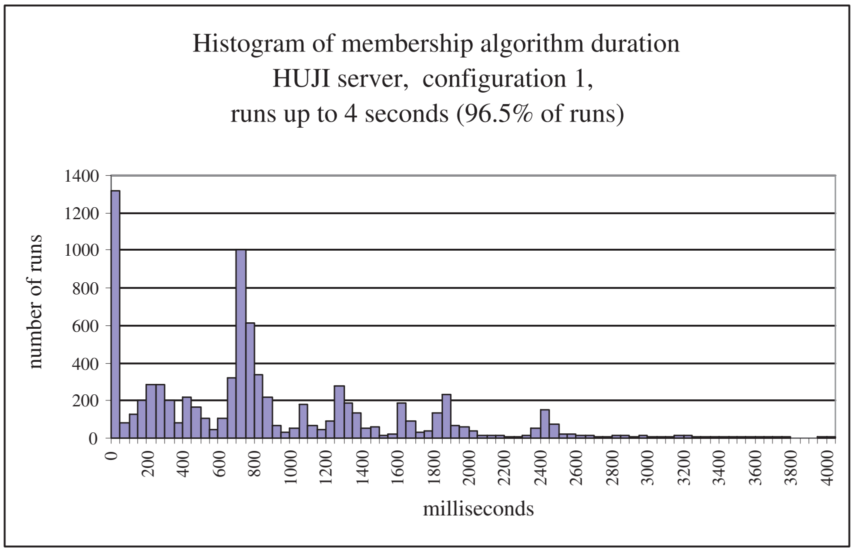


Fig. 16. A histogram of Moshe duration at HUJI, first configuration.

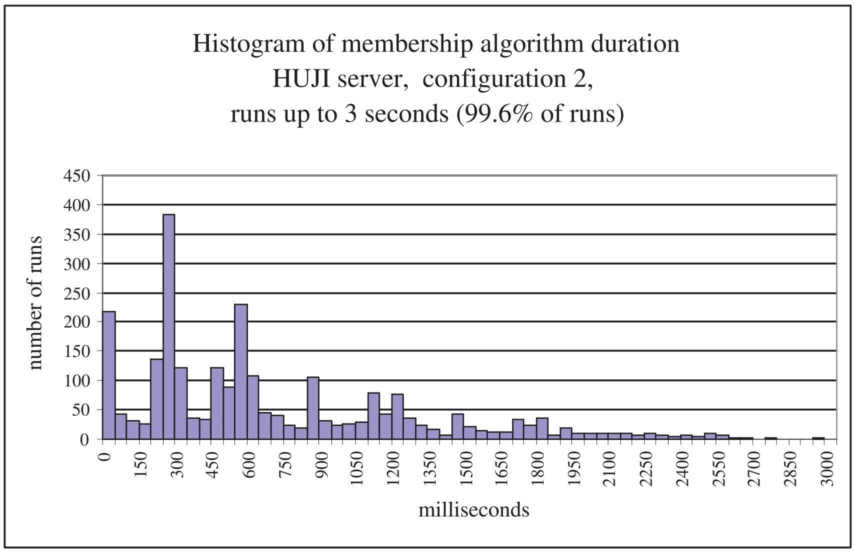


Fig. 17. A histogram of Moshe duration at HUJI, second configuration.

and MIT reach HUJI in 300 ms. Also ignore local computation time. If a client at CU joins a group, a `NE` *reflecting this join is reported to the MIT server as fast as 10 ms. later, whereas at HUJI, it is reported only 300 ms. after the join. Thus, the server at MIT invokes Moshe 290 ms. before the HUJI server. A proposal from MIT reaches HUJI 10 ms. after the* `NE`*, and at this point the HUJI server can deliver the view. The proposal from HUJI, on the other hand, is only sent to MIT after* *the* `NE` *at HUJI, and reaches MIT 300 ms. later, which is 590 ms. after Moshe is invoked by the MIT server. In this example, the duration of Moshe at HUJI is 10 ms. whereas at MIT it is 590 ms.*

This example is representative for invocations of Moshe due to an event at one of the US sites, when the NTU server is not involved in the view. This accounts for only some of the cases in the first peak in Figures 16 and 17. When the NTU server is involved, the typical duration at HUJI for views initiated in the US goes up to around 300 ms., due to the time it takes the join report to reach NTU, plus the time it takes the proposal from NTU to reach HUJI. If the join report to HUJI is delayed (due to loss), then the duration of Moshe at HUJI becomes even shorter while at the other locations it becomes longer. We believe that such loss accounts for most of the cases in the first peak. In the second configuration, when the loss rate was lower, the first peak was smaller.

About one fifth of the join and leave events were generated at HUJI. These behave conversely to the cases in the example. At MIT, cases initiated at HUJI and NTU often terminate quickly, especially if not all of the servers are involved in the view. This accounts for the small peak around 250 ms. in Figures 11 and 14 above.

It is worth noting that the difference in the starting times of Moshe in different locations is an artifact of running on a WAN, where latencies among different processes are disperse. This does not stem from any design decisions made in our algorithm; no algorithm can be initiated at a remote site before that site learns that the algorithm should be initiated, and a remote site cannot learn that the algorithm should be initiated before it receives a message that causally follows the initiating event at the origin.

8. PROVIDING VIRTUAL SYNCHRONY

Moshe is designed to be used in conjunction with a group 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, for a survey, see [Chockler et al. 2001]. 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 Moshe. A deeper discussion can be found in [Keidar and Khazan 2000].

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 it was the default initial view V_0 if no view had yet been 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 (for example, *strong virtual synchrony* [Friedman and van Renesse 1995], and the specifications of [Fekete et al. 2001; Moser et al. 1994; Keidar and Khazan 2000]) support a stronger property called *Sending View Delivery*, which ensures 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 that processes moving together from view $V1$ to view $V2$ deliver the same set of messages in $V1$. In order to exploit this property, a process moving from view $V1$ to view $V2$ needs to know who are the other member that also continue directly from $V1$ to $V2$. This information is conveyed to the client along with the view, it is often called the *transitional set* [Moser et al. 1994; Chockler et al. 2001].

Virtual synchrony properties are implemented by synchronizing participating processes while view changes are taking place (for examples, see [Friedman and van Renesse 1995; Guo et al. 1996; Agarwal et al. 1998; Keidar and Khazan 2000]). 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. When view changes are taking place, clients send each other special synchronization messages in order to agree upon the set of messages they will deliver in the old view before moving to the next one.

Moshe 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 synchronization message to the other clients, tagged with the `startChangeNum` of the `startChange` message. The synchronization message also carries the information required to agree on the set of the messages to be delivered in the view that is now ending, as well as the identifier of the view that is now ending. If Sending View Delivery is desired, then the client blocks the sending of messages after sending a synchronization message until the next view is delivered.

When a client receives a `view` message V from its server, the client needs to ensure that it delivers the same set of messages as other clients before delivering V to its application. To this end, the client collects synchronization messages from all of the clients that continue with it from the current view to V. Clients use the information in the synchronization 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 synchronization 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 synchronization message to be used from c. The clients use the identifier of the previous view included in the synchronization messages to compute the transitional set.

9. RELATED WORK

We have described Moshe, a one-round membership algorithm and service for wide-area networks. We now compare our service with related work.

9.1 One round membership

Nearly all previous virtually synchronous group membership algorithms are two-round algorithms. For example, the algorithms employed in Isis [Ricciardi and Birman 1991], Horus [van Renesse et al. 1994], Ensemble [Hayden and van Renesse 1996], Relacs [Babaoğlu et al. 2001], Transis [Dolev et al. 1994], Totem [Agarwal et al. 1998], and Phoenix [Malloth et al. 1995], all perform two communication rounds after the network stabilizes and all of the participates know the correct network situation.

Some algorithms, e.g., those of [Hiltunen and Schlichting 1998; Cristian and Schmuck 1995], organize process in a logical ring structure, and can have the membership algorithm terminate after the token circulates the ring twice. The first iteration is used to propagate information about locally detected connectivity changes, and the second, to agree on the membership. Each iteration takes as many communication steps as the number of processes in the system. Therefore, this approach cannot work effectively in a WAN where communication steps are costly. As a consequence of using a ring structure to propagate information about connectivity changes, the information is already ordered once it has propagated, that is, processes cannot differ in the order in which they perceive connectivity changes. This eliminates the types of scenarios that lead to running the slow agreement algorithm in Moshe. In other words, the first iteration does more than the notification service used by Moshe; it orders the information in addition to propagating it.

The only other single round membership algorithm that we are aware of is the one-round membership algorithm in [Cristian and Schmuck 1995]. 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. This membership algorithm is not suitable for WANs, where δ tends to be big and typical network events are partitions and merges. In contrast, 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. In our experiments, the algorithm terminated within one round, (i.e., δ time) in almost 99% of the cases.

The Optimistic Atomic Broadcast algorithm of [Pedone and Schiper 1998] has a structure very much like the single-round/three-round structure of Moshe. In both cases, the algorithms are optimized to perform quickly when events are well ordered (for Moshe when a network event arrives at the appropriate servers, and for Optimistic Atomic Broadcast when the messages already arrive in a total order).

9.2 Separating membership maintenance from multicast services

Following the approach taken by CONGRESS [Anker et al. 1997] and Maestro [Birman et al. 1998], our design separates the maintenance of membership from the group multicast: membership is not maintained by every group member but only by dedicated membership servers that 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 [Keidar and Khazan 2000]. Unlike Maestro, 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.

9.3 Group communication services for WANs

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 [Dolev and Malkhi 1996; Agarwal et al. 1998; Amir and Stanton 1998; Rodrigues and Verissimo 2000]. These systems leverage off of the fact that WANs are interconnected LANs. The membership algorithms implemented in such systems usually first run the original membership 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 [Agarwal et al. 1998] takes two rounds per ring[6] plus an extra round for multiple rings [Agarwal et al. 1998].

Our algorithm 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.

9.4 Light-weight group membership services

Light-weight group membership services (for example, [Dolev and Malkhi 1996; Amir and Stanton 1998; Powell 1991; Glade et al. 1993; Rodrigues et al. 1996; Birman et al. 1998]) 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, for example, those of [Dolev and Malkhi 1996; Amir and Stanton 1998; Powell 1991; Glade et al. 1993], 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, Moshe only computes the process-level group membership, hence additional message rounds for computing both memberships are not necessary. Furthermore, Moshe provides clients with full virtual synchrony semantics.

Light-weight group membership services scale well in the number of groups maintained, since they maintain the membership for several groups at the same time, and can therefore bundle together messages pertaining to membership changes in different groups. It is possible to implement a similar optimization in Moshe, since in our design, the same membership servers maintain the membership of all of the groups. Thus, it should not be difficult to modify Moshe servers to also handle membership changes concerning several groups at the same time, and to bundle messages corresponding to different groups into a single message.

Thus, Moshe 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.

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

10. CONCLUSIONS

We have described Moshe, a group membership algorithm for wide-area networks. We have proven that Moshe provides properties that are useful and attainable in an asynchronous system that may suffer communication failures and partitions, but eventually stabilize.

We have ran Moshe over the Internet for almost two weeks, during which the algorithm delivered over 12,250 membership views. We experimented with two different configurations. Our experiments led to interesting general observations regarding the behavior of membership algorithms over the Internet. The experiments also illustrated the utility of Moshe's features:

(1) Moshe does not deliver obsolete views to its clients. Obsolete views arise from instability in the network. By not delivering obsolete views, Moshe reduces the overhead of virtual synchrony: applications need not handle view changes to views that no longer exist. Moreover, during periods of instability in the network, Moshe does not generate additional traffic which could exacerbate the instability. In our experiments, instability lasted over 20 seconds in only 1.5% of the cases in one configuration and in merely 0.35% of the cases in the other configuration.

(2) Moshe optimizes for the common case of the failure detection being relatively consistent. This occurred in nearly 99% of the view changes in one configuration and in 99.8% in the other configuration.

(3) Moshe is built on top of a network event notification service. One can configure the underlying service to optimize for different network conditions. We have seen that the configuration of the notification service has a major effect on the performance of Moshe. By abstracting the notification service out we could design a simple algorithm that works the same way in all configurations.

(4) Moshe is built with a client-server design in which the membership is not maintained by every process, but only by dedicated membership servers.

We have validated the utility of the fourth feature with a set of experiments presented elsewhere [Keidar et al. 2000]. The experiments were run using a prototype notification service before CONGRESS was available. They indicate that Moshe should easily scale to systems containing hundreds of clients. These experiments were quite straightforward and the results were not surprising; introducing a hierarchy is a well-known technique for achieving scalability (see, for example, [Guo et al. 1996]). Therefore, we did not reproduce these results here.

Acknowledgments

We are thankful to Tal Anker, Gregory Chockler, Alan Fekete, Roger Khazan, and Ohad Rodeh for many interesting discussions and helpful suggestions. We thank Omri Barel and Tal Anker for providing us with the CONGRESS implementation used in our tests. We thank Yuh-Jzer Joung and Robbert van Renesse for hosting our experiments on their machines in Taiwan and Cornell University, respectively. Finally, we thank the anonymous referees for their helpful comments which helped us improve the quality of the presentation.

REFERENCES

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

Agarwal, D. A., Moser, L. E., Melliar-Smith, P. M., and Budhia, R. K. 1998. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems 16*, 2 (May), 93–132.

Amir, Y., Dolev, D., Melliar-Smith, P. M., and Moser, L. E. 1994. Robust and efficient replication using group communication. Technical Report CS94-20, Institute of Computer Science, Hebrew University, Jerusalem, Israel.

Amir, Y. and Stanton, J. 1998. The Spread wide area group communication system. TR CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University.

Andersen, D. G., Balakrishnan, H., Kaashoek, F., and Morris, R. 2001. Resilient overlay networks. In *SOSP* (Oct. 2001).

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

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

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

Babaoğlu, Ö., Davoli, R., and Montresor, A. 2001. Group communication in partitionable systems: Specification and algorithms. *IEEE Trans. Softw. Eng. 27*, 4 (April), 308–336. Previous version: University of Bologna Department of Computer Science Technical Report UBLCS98-1.

Birman, K. 1996. *Building Secure and Reliable Network Applications*. Manning.

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

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

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

Chockler, G. V., Keidar, I., and Vitenberg, R. 2001. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys 33*, 4 (December), 1–43. Previous version: MIT Technical Report MIT-LCS-TR-790, September 1999.

Cristian, F. and Schmuck, F. 1995. Agreeing on process group membership in asynchronous distributed systems. Technical Report CSE95-428, Department of Computer Science and Engineering, University of California, San Diego.

Dolev, D. and Malkhi, D. 1996. The Transis approach to high availability cluster communication. *Commun. ACM 39*, 4 (April), 64–70.

Dolev, D., Malki, D., and Strong, H. R. 1994. An asynchronous membership protocol that tolerates partitions. Technical Report CS94-6, Institute of Computer Science, Hebrew University, Jerusalem, Israel.

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

Fekete, A., Lynch, N., and Shvartsman, A. 2001. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems 19*, 2 (May), 171–216. Previous version appeared in PODC 1997.

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

Friedman, R. and Vaysburg, A. 1997. Fast replicated state machines over partitionable networks. In *16th IEEE International Symposium on Reliable Distributed Systems (SRDS)* (October 1997).

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

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

Guerraoui, R. and Schiper, A. 1997b. Software-based replication for fault tolerance. *IEEE Computer 30*, 4 (April), 68–74.

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

Hayden, M. and van Renesse, R. 1996. Optimizing layered communication protocols. Technical Report TR96-1613 (November), Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA.

Hiltunen, M. A. and Schlichting, R. D. 1998. A configurable membership service. *IEEE Transactions on Computers 47*, 5 (May), 573–586.

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

Keidar, I. and Khazan, R. 2000. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20th International Conference on Distributed Computing Systems (ICDCS)* (April 2000), pp. 344–355. IEEE Computer Society Press. Full version: MIT Lab. for Computer Science Tech. Report MIT-LCS-TR-794.

Keidar, I., Sussman, J., Marzullo, K., and Dolev, D. 2000. A client-server oriented algorithm for virtually synchronous group membership in WANs. In *20th International Conference on Distributed Computing Systems (ICDCS)* (April 2000), pp. 356–365. Full version: MIT Technical Memorandum MIT-LCS-TM-593a, June 1999, revised September 2000.

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

Malloth, C. P., Felber, P., Schiper, A., and Wilhelm, U. 1995. Phoenix: A toolkit for building fault-tolerant, distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products* (October 1995).

Mishra, S., Peterson, L. L., and Schlichting, R. L. 1993. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal 1*, 2 (December), 87–103.

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

Pedone, F. and Schiper, A. 1998. Optimistic atomic broadcast. In *12th International Symposium on DIStributed Computing (DISC)* (September 1998), pp. 318–332.

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

Ricciardi, A. M. and Birman, K. P. 1991. Using process groups to implement failure detection in asynchronous environments. In *ACM Symposium on Principles of Distributed Computing (PODC)* (August 1991), pp. 341–352.

Rodrigues, L., Guo, K., Sargento, A., van Renesse, R., Glade, B., Verissimo, P., and Birman, K. 1996. A dynamic light-weight group service. In *15th IEEE International*

Symposium on Reliable Distributed Systems (SRDS) (Oct. 1996), pp. 23–25. also Cornell University Technical Report, TR96-1611, August, 1996.

Rodrigues, L. and Verissimo, P. 2000. Topology-aware algorithms for large-scale communication. In S. Krakowiak and S. Shrivastava Eds., *Advances in Distributed Systems*, Volume 1752 of *LNCS* (2000). Springer Verlag.

Schiper, A. and Ricciardi, A. 1993. Virtually synchronous communication based on a weak failure suspector. In *23rd IEEE Fault-Tolerant Computing Symposium (FTCS)* (June 1993), pp. 534–543.

Sussman, J., Keidar, I., and Marzullo, K. 2000. Optimistic virtual synchrony. In *19th IEEE International Symposium on Reliable Distributed Systems (SRDS)* (October 2000), pp. 42–51.

Sussman, J. and Marzullo, K. 1998. 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.

van Renesse, R., Hickey, T. M., and Birman, K. P. 1994. Design and performance of Horus: A lightweight group communications system. TR 94-1442 (August), dept. of Computer Science, Cornell University.

van Renesse, R., Minsky, Y., and Hayden, M. 1998. A gossip-style failure detection service. TR TR98-1687 (May), Cornell University, Computer Science.

APPENDIX

A. CORRECTNESS OF THE MEMBERSHIP ALGORITHM

We prove here that the algorithm fulfills the properties specified in Section 4. In Section A.1 we prove that it fulfills the client interface properties: **Monotonicity of startChange Identifiers** and **Integrity of startChange Identifiers**. In Section A.2, we prove that the algorithm satisfies the membership properties: **View Identifier Local Monotonicity** and **Agreement on Views**.

A.1 Client interface properties

Proposition A.1. *(Monotonicity of startChange Identifiers)* `startChange` *identifiers sent to each client are monotonically increasing.*

Proof. Whenever a `startChange` message is sent to the clients, (Figure 6, NE event handler), `startChangeNum` is first increased and then sent in the message. □

Proposition A.2. *(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]`= $SM.startChangeNum$ *and* V.`members` *is equal to* SM.`suggestedMembers`.

Proof. A server s sends its clients two types of messages: `view` and `startChange`. Whenever a `startChange` message is sent, (Figure 6, NE event handler), 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, (Figure 6, proposal event handler) the server checks that `props` contains `proposal` messages from all of the servers of members of the view, including itself. The new `view` to be sent is stored in `curView`, and `curView.startChangeNums[s]` is selected to be `props[s].startChangeNum`, which contains the latest value of `startChange.startChangeNum` sent to local members of the view. Furthermore, every time a `NE` occurs, the server sends a new `startChange` message to the client with `suggestedMembers` equal to the up-to-date `NSView`. Since a server only delivers views that match its `NSView`, V.`members` is always equal to the `suggestedMembers` sent in the latest `startChange` message.

Upon sending a `view`, the server removes this `proposal` messages from `props` by setting `props[s]` to null. 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. □

A.2 Membership properties

PROPOSITION A.3. *(View Identifier Local Monotonicity) 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 (Figure 6, proposal event handler), $V.id$ 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, (Figure 6, NE event handler), $startChangeNum$ is chosen to be greater than `curView.id`. By Proposition A.2, at least one `startChange` event is sent to clients between $V1$ and $V2$. The proof follows. □

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 A.4. *(Fast Agreement Blocking Detection) If the fast agreement algorithm does not terminate successfully, then the detection mechanism (described in Figure 7) detects the blocking after time t_0.*

PROOF. As above, we denote by $last_s$ the last `proposal` message of type `FA` sent by a server $s \in SS$. Since these proposals are sent in response to the last `NE` at each server, for every $s \in SS$, $last_s.members$ is exactly CS.

If every server $s \in SS$ uses exactly the proposals in the set $\{last_{s'} | s' \in SS\}$ for a view, then all the servers in SS send the same view to their clients, this view is sent after the last `startChange`, and it correctly reflects the network situation. That is, in this case, the fast agreement algorithm terminates successfully. Therefore, if the fast agreement algorithm does not terminate at some server in SS, it must be the case that not all the servers in SS use exactly the proposals in the set $\{last'_s | s' \in SS\}$ for a view. By the **Reliable Links** property, s receives all the proposals in this set. Moreover, as noted above, all the proposals in this set propose the same membership. Therefore, if s does not use them together for a view, it must be the case that s already used at least one of these proposals for an earlier view before receiving all of them. We distinguish between two cases:

(1) There exists a server $s' \in SS$ such that for some view V, s uses $last_s$ but not $last_{s'}$. Since $last_s$ contains clients of s' (by definition of SS), s uses some earlier `proposal` message from s' for V.

(2) There exists a server $s' \in SS$ such that for some view V, s uses $last_{s'}$ but not $last_s$. In this case, s uses an earlier `proposal` message of its own for V.

We now prove that both of these cases will result in detections, in other words, the function `TestIfSAProposalNeeded` (Figure 7) at one of the servers will return TRUE. If `running` is already `SA` at one of the servers, then the block has been detected and we are done. Assume for the rest of this lemma that `running` is not `SA` at any of the servers.

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 (Property FIFO **Order**), all `proposal` messages from s' received by s are received in the order they are sent. Thus, $last_{s'}$ is 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` is `none`, and this results 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 $last_{s'}$ (Figure 6, proposal event handler). 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' (Figure 7, `TestIfSAProposalNeeded`). □

LEMMA A.5. *(No False Blocking Detection) The detection mechanism described in Figure 7 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, that is, `TestIfSAProposalNeeded` (Figure 7) 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 send a view V using $last_{s'}$ for every s' in SS. Before s sends $last_s$, the `NSView` at s is not CS, so a $last_{s'}$ received by s before it sends $last_s$ will not result in detection (Figure 6, proposal event handler does nothing for proposals that do not match `NSView`). 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 $last_s$ (Figure 7, `TestIfSAProposalNeeded`).

The `usedProps` function of s' is updated before s' sends a `view` to its clients (Figure 6, proposal event handler). 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 successfully. □

LEMMA A.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 algorithm is invoked by some server s in SS, then eventually every server s' in SS will enter the slow agreement algorithm by sending a proposal of type SA, (Figure 7, TestIfSAProposalNeeded). 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 algorithm by a server s has a greater propNum than any proposal of type SA received by s beforehand (Figure 7, SendSAProposal).

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 its last NE. 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 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 is exactly the set 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 A.4, the slow agreement algorithm will be run. The slow agreement algorithm always terminates, as proven in Lemma A.6.

It remains to be proven 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 A.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 A.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. □