DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

Scalable Group Membership Services for Novel Applications

Tal Anker, Gregory V. Chockler, Danny Dolev, and Idit Keidar

ABSTRACT. Group communication is a useful abstraction in the development of highly available distributed and communication-oriented applications in wide area networks (WANs). The most important aspects of this abstraction are the dynamic maintenance of group membership and its diverse semantics for interleaving membership change notifications within the flow of regular messages.

In this paper we propose a new architecture for a scalable group membership service for wide area environments. Our architecture provides two different service levels and their semantics, each geared to different applications with different needs: The CONGRESS membership service which provides simple semantics of membership approximation, and the MOSHE service, which extends CONGRESS, provides full virtual synchrony semantics.

The novelty of our design is in its client-server approach, which allows lightweight clients to benefit from advanced membership services. Furthermore, our design supports the coexistence of full-fledged clients along with thin clients.

1. Introduction

Group communication [**ACM96**] is an important abstraction, widely used for distributed and communication-oriented applications. Such applications typically require the coordination of large and dynamic sets of processes at different sites. The group communication abstraction is essential for the modular design of groupware and other multi-user applications in such networks. The most important aspects of this abstraction are the maintenance of group membership and the semantics of interleaving membership change notifications within the flow of regular messages.

Different applications utilize group communication for different purposes, and hence require different semantics from the group membership service they utilize (as explained in [**BFHR98, CHKD96, Bir96**]). For example, video conferencing applications need a general knowledge of which peers are interested in joining the conference, in order to know where to multicast the video stream, and where to receive it from. Such applications do not require the synchronization of membership

1991 *Mathematics Subject Classification.* Primary 68-06, 68M10; Secondary 90B12, 90B25.

This work was supported in part by ARPA grant #030-7310 and by the Israeli Ministry of Science grant #032-7658.

Idit Keidar's research was supported by the Israeli Ministry of Science.

change notifications with regular messages. Other examples are pay-TV and highly-available video-on-demand servers [**ACK+97**].

On the other end of the spectrum, consistent data replication may greatly benefit from strong semantics [**BJ87, ABCD96, KD96, FLS97, ADMSM94, SM98**]. For example, some group communication systems provide virtual synchrony semantics, which synchronize membership notifications with regular messages and thus simulate a "benign" world in which message delivery is reliable within the set of live processes. This enables synchronization among applications, but is costly: it incurs a delay period in which messages may not be transmitted [**FvR95**]. Therefore, it is not appropriate for applications that require real-time message delivery (e.g., video transmission).

Computer Supported Cooperative Work (CSCW) [**Rod91**] groupware and multimedia applications involve different services that require different Qualities of Service (QoS) and different semantics from the group membership which they use, for example, an on-line conferencing application may incorporate multimedia multicast as well as coordination and sharing of consistent information (e.g., a shared white board).

Extensive research is currently being carried out to optimize scalable reliable multicast protocols in order to meet the demands of such applications [**Car94, FJM+95, PSK94, PSLB97**]. Many of these applications make use of highly dynamic *multicast groups*. One example is a TV broadcasting service that serves groups of clients that may join or leave at any time. Such protocols often need to be complemented by a membership mechanism that maintains the dynamically changing set of members in each multicast group.

However, the design of a scalable membership service for WANs is a challenging task. A typical multicast group over a WAN may consist of a large number of members, which may be geographically far apart. These conditions cause membership to be highly dynamic. A protocol that manages the membership information will be forced to propagate large amounts of membership data across long distances.

In this paper we describe a new architecture for construction of a scalable group membership service for wide area environments. Our membership service provides two different service levels and semantics, each geared to different applications with different needs. In addition, our membership server provides advanced services such as a hierarchical directory of groups and secure group services.

The two different service semantics are geared towards different kinds of applications: the CONGRESS [**ABDL97, Ank97**] membership service provides simple semantics of membership approximation, and the MOSHE [**KSDM**] service, which extends CONGRESS, provides full virtual synchrony semantics.

In our design, membership is not maintained by every process, but only by a few dedicated servers. Thus, membership maintenance induces very low overhead when membership changes are infrequent, and the strong semantics required by some parts of the application induce no overhead for those parts which require weaker semantics.

In order to be scalable and efficient, our service minimizes the network traffic required to maintain the dynamic group membership. The saving in network traffic is achieved by using a hierarchy of dedicated servers, which propagate necessary information about multicast groups to clients in the server's area.

The rest of this paper is organized as follows: In Section 2 we describe the environment model and in Section 3 we describe the system architecture. The basic membership service, CONGRESS, is described in Section 4, and the virtually synchronous membership service, MOSHE, in Section 5. In Section 6 we describe the advanced membership services that we provide. Section 7 concludes the paper.

2. The environment model

Processes communicate by exchanging messages. There is no bound on message delivery time, hence the system is asynchronous. Processes fail by crashing, and crashed processes may later recover. Live processes are considered *correct*, crashed processes are *faulty*.

Communication links may fail. A *connected component* C is a set of correct processes among which all the communication links are operational, and all the links from processes in C to processes outside C are not operational. The system is equipped with a (possibly unreliable) external failure detector module, which is described in Section 2.2, and reliable FIFO communication links, which are described in Section 2.1.

2.1. Reliable FIFO links. The communication links between pairs of processes preserve the FIFO order, i.e., if process p sends two messages to process q: m_1 and later m_2, and if q delivers both messages, then these messages are delivered in the order in which they were sent.

Furthermore, while process q does not suspect process p, p's messages are delivered to q without gaps, i.e., if process p sends a message m' between m_1 and m_2, and between the delivery of m_1 and m_2, q does not suspect p, then q also delivers m' between m_1 and m_2.

2.2. Failure detector. Our membership server exploits an external *failure detector (FD)* module which reports changes in the network topology and in process liveness. We say that a process p *suspects* a process q if the failure detector module at p reports q as faulty/disconnected. The failure detector may be inaccurate, i.e., it may suspect correct processes. However, we assume that the failure detector is *complete*, i.e., it eventually suspects every process that has permanently crashed or disconnected.

We assume that the failure detector module operates in conjunction with the underlying communication, so that no messages are ever delivered from a suspected process. In this paper, we do not discuss how the failure detector is implemented. A framework for implementing a failure detector in a WAN is provided in [**Vog96**]. Theoretical aspects of failure detectors are discussed in [**CT96, DFKM96, DFKM97**].

3. The service architecture

Group communication systems introduce the notion of group abstraction, which allows processes to be easily arranged into multicast groups. Each message targeted to the group is delivered to all currently connected and operational group members. Group communication systems typically support reliable group multicast and membership services. The task of the membership service is to maintain a listing of the currently active and connected group members, and to deliver this information to the application whenever it changes.

Our membership service differs from those of other group communication systems in that it complies with the client-server paradigm (please see Figure 1).

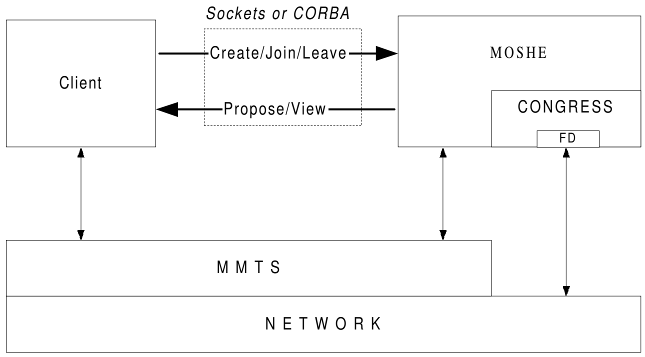

FIGURE 1. The membership service architecture.

Processes that communicate with each other are *clients* of the membership service. The clients communicate with each other using a multicast transport service, called MMTS [**CHKD96**] which allows them to multicast messages to all the members of a group. In this section, we describe the overall design of the membership service. The description of the multicast service is beyond the scope of this paper.

In Section 3.1 we explain how client and server communicate. The membership server structure is described in Section 3.2. The membership server uses an external failure detector, as described in Section 2.2.

3.1. Client-Server interaction. The membership server interface recognizes the following events:

Create: A request to create a group.
Join: A request to join a group.
Leave: A request to leave a group.
Client Failure: A report of a client failure.

DEFINITION 3.1. We say that a process p is a *member* of a group G in a connected component C, if p is currently in C, p has joined G (i.e., issued a join), and afterwards p has neither crashed nor issued a leave .

The client interface recognizes the following events:

Propose: A report of the beginning of the synchronization phase (see below).
View: A report of a new group membership.
Server Failure: A report of server failure.

We currently support two options for client-server interaction: The first option is based on a reliable point-to-point FIFO service built directly atop the low-level socket interface. The second option utilizes the *Common Object Request Broker Architecture (CORBA)* which is the industrial standard for building client-server applications. Some of the advantages of using CORBA include: simplified object-oriented design, network transparency, client-server failure detection, and the possibility of using standard CORBA services (e.g., security, naming and event services).

Within the CORBA framework, objects (i.e., entities consisting of an interface and an implementation) are *registered* over a virtual *software bus*, called the *Object Request Broker (ORB)*. Whenever a CORBA application issues a request to

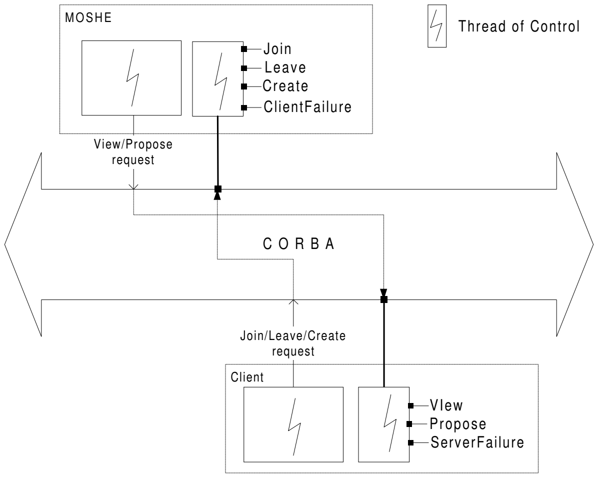

FIGURE 2. The client-server interaction using CORBA.

a previously registered (possibly remote) object, the ORB locates the object and forwards the request to it.

The membership client-server interaction using CORBA is illustrated in Figure 2. For the membership server, an object implementing the Create/Join/Leave interface is registered over the ORB. For the membership client, an object implementing the Propose/View interface is also registered over the ORB. Both objects run in separate threads. Additionally, both objects implement methods for handling client/server failures. Whenever a membership client (server) wishes to invoke some server's (client's) method, it simply makes a regular function call in its address space. All remaining steps (which include: locating the remote object, call parameters marshalling/de-marshalling and message passing) are transparently accomplished by the ORB.

3.2. The membership server. The membership server is designed according to the object-oriented paradigm and written in the Java programming language [**Now98**]. The membership server consists of two objects: the *Membership Object-oriented Service for Heterogeneous Environments (*MOSHE*)* and *CONnection-oriented Group-address RESolution Service (*CONGRESS*)*. The CONGRESS substratum accumulates the group membership and failure detection information and disseminates it among the membership servers. CONGRESS resides directly on top of a network layer (such as ATM or IP).

MOSHE extends CONGRESS to provide membership services with strong membership and message delivery semantics. Examples of such semantics include virtual synchrony and the ordered delivery of views. In addition, MOSHE provides some advanced services such as hierarchical and secure group services. In order to synchronize multicast message delivery with membership change events, MOSHE clients multicast synchronization messages via the MMTS.

Client requests are first processed by the MOSHE object. For each client request, this processing includes updating the group hierarchy (if necessary) as well as authorization and/or authentication for secure groups. Then, the request is delegated to the CONGRESS object for further dissemination among the other membership servers.

When a change in the membership of some group is reported by CONGRESS, MOSHE checks if the group requires strong semantics. If it doesn't, MOSHE immediately informs the group members of the new membership. Otherwise, MOSHE initiates an additional synchronization round at the end of which the view is reported.

4. CONGRESS

Maintenance of a multicast group membership in a WAN environment is a challenge. The membership of the multicast groups may change dynamically and the multicast groups themselves may be created and disposed of on-demand. Failure detection is harder in a WAN environment, and the probability of failures is higher. Thus a protocol for *resolving* (i.e., mapping a multicast group name into a set of members identifiers) and maintaining the membership of multicast groups geared to the WAN is necessary. CONGRESS is such a protocol.

4.1. CONGRESS design principles.

CONGRESS is a protocol for maintaining multicast group membership information. CONGRESS operates over point-to-point connections, and is scalable to a WAN. It should be noted that CONGRESS is not designed to deal with the actual application data transfer within a multicast group. CONGRESS separates the multicast group management from the data transfer issues. Thus, maintenance of multicast group membership does not affect the QoS provided by the network.

CONGRESS' design is based on the following principles:

- **No flooding:** CONGRESS does not flood the WAN on every group membership change. This is achieved through careful maintenance of a distributed spanning tree for each of the multicast groups. A single membership change in a multicast group G incurs $O(|G|)$ protocol messages.
- **Hierarchical design:** CONGRESS services are provided to applications by hierarchically organized servers. The hierarchical design minimizes the size of the data structures maintained by each server. CONGRESS defines *scopes* within which group membership information should be propagated in order to conserve network resources.
- **Robustness:** CONGRESS is designed to operate in a WAN, where there is a greater possibility that CONGRESS servers may temporarily disconnect and later reconnect, due to network failures or network reconfiguration and re-planning. CONGRESS withstands such transient errors by providing a best-effort service. Applications continue to receive the CONGRESS services from those servers to which they remain connected. The small amount of data maintained by the servers allows for simple recovery.

As was previously noted, a scalable multicast group membership service is very important for multicast oriented applications. CONGRESS provides for decentralization, load sharing, and fault tolerance.

4.2. CONGRESS services. The CONGRESS services are provided by an interface that consists of the following basic functions:

- **join**(G): Make the invoking client a registered member of group G.
- **leave**(G): Unregister the invoking member from G.
- **resolve**(G): Request to resolve a multicast group name G into an approximated set of members identifiers.
- **set_flag**(G, $online\_flag$): Enable or disable the reception of the *incremental membership notifications* w.r.t. G, by the invoking member.

A client may learn of the membership of a group in one of two ways:

- **resolve-reply** is a response to a *resolve* request. It consists of an approximated list of members.
- **Incremental Membership Notification** is a notification that reflects a change in the group's membership, due to *join*, *leave*, *process/communication link* failure. Incremental membership notifications reflect only the difference between the new membership and the previously reported one.

The *membership of group G computed by a user* is constructed by resolving a group name once, and subsequently applying the incremental membership notifications as they arrive.

4.3. Overview of the CONGRESS architecture. CONGRESS services are provided by a set of *servers*. There are two kinds of CONGRESS servers: *Local Membership Servers (LMSs)* and *Global Membership Servers (GMSs)*. An LMS resides in each host and is a CONGRESS front-end for the clients on that host. GMSs are organized in a hierarchical structure throughout the network, and may run either on dedicated machines or in routers. GMSs maintain data structures that efficiently aggregate the global membership information. Neighboring servers communicate via reliable FIFO links.

Failure handling in CONGRESS focuses on asynchronous host crash/recoveries, and communication link failures/recoveries. In order to handle these failures, each CONGRESS server interacts with the local failure detector module that monitors the liveliness of the particular CONGRESS server's neighbors.

CONGRESS views the network as a hierarchy of *domains*, where each domain is serviced by a CONGRESS server. At the lowest level, a domain consists of a set of clients (applications) running on the same host. Such domains are called *host domains*. Each host domain is serviced by the LMS of its respective host. The LMS is called a *representative* of a host domain. Higher level domains consist of a set of lower level domain representatives. Thus, a single GMS may serve a domain that consists of either several LMSs, or several GMSs that are representatives of their respective lower level domains.

A CONGRESS *domain identifier* is the longest common address prefix of the lower level domains which make up a particular domain. The domain identifier of a host-domain is the full address of the LMS' host. An example is shown in Figure 3: Here, one can see that the sample domain 1.1 is comprised of two domains: 1.1.1 and 1.1.2. The domain 1.1.2 is comprised of three host-domains 1.1.2.1, 1.1.2.3 and 1.1.2.4.

In order to avoid flooding the whole network with every membership change in every group, membership notifications pertaining to a specific multicast group are propagated using a *distributed spanning tree* for this group. This spanning tree is a

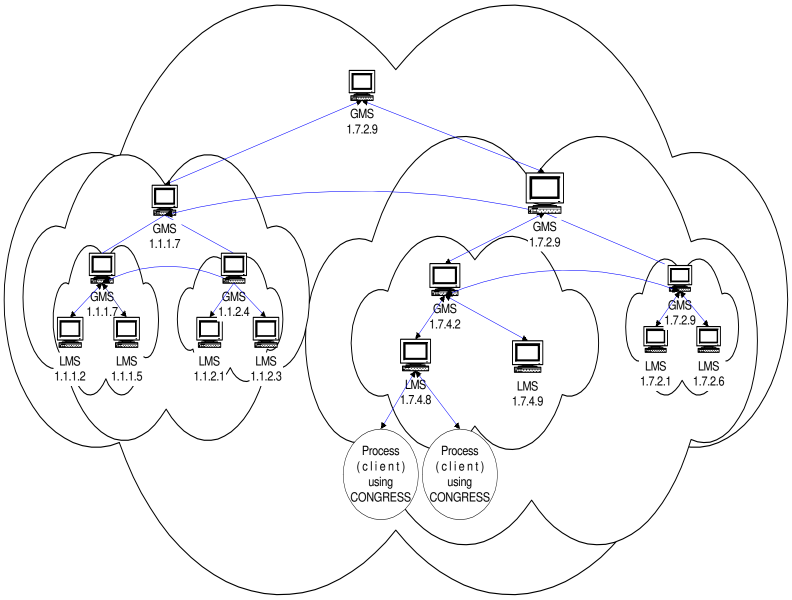

FIGURE 3. The CONGRESS hierarchy structure.

sub-tree of the CONGRESS servers hierarchy. The CONGRESS servers comprising the sub-tree of a group, are those which have members of this group in their domains. Each server in the CONGRESS hierarchy maintains only that part of the spanning tree which consists of its immediate neighbors. The spanning tree is constructed and maintained according to member join/leave requests. In addition, network topology changes and client or server crash/recovery events[1] change the topology of the spanning tree. Obviously, since CONGRESS operates in an asynchronous environment, the spanning tree of a group can only be a best-effort approximation.

The details of the CONGRESS algorithm may be found in [**ABDL96, ABDL97**].

4.4. CONGRESS **guarantees.** In this section, we intuitively describe the properties that CONGRESS guarantees w.r.t. the membership information it provides.

Membership updates and replies to resolve requests are propagated by CONGRESS and received by its clients in an order that reflects that of the membership events' (or membership changes) occurrence. This enables a client to construct an up-to-date view of the group membership based either upon membership updates or upon resolve replies. The ordering of membership events that CONGRESS guarantees is defined only w.r.t. events that involve a particular host, domain or a client. This guarantee is called *per-source chronological ordering of membership events.*

PROPERTY 4.1 (Per-Source Chronological Ordering of Membership Events). *Any two membership events that involve the same source will be reported to every member in the order that they occurred.*

[1] Such events are detected by the *failure detector* module.

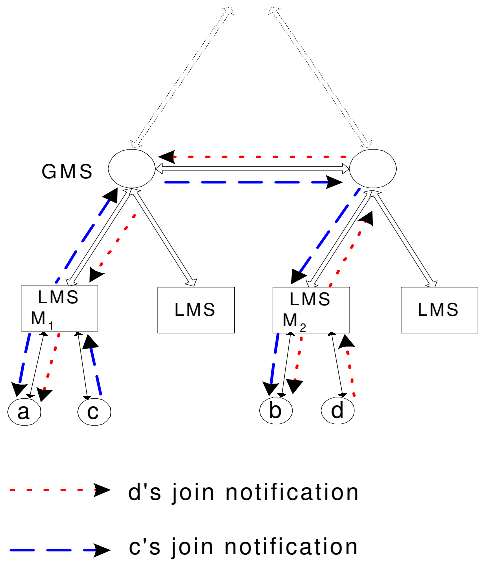

FIGURE 4. Two clients concurrently join a CONGRESS group.

Note, however, that CONGRESS may report membership events (corresponding to different sources) in different orders at different sites. This is illustrated in Example 4.1 below. Agreement on the order of notifications would require running a synchronization round for each membership change. This synchronization round is performed by MOSHE for groups that require it, as explained in Section 5.

EXAMPLE 4.1. Assume that a and b are two members of a group G. Assume further that the membership at both of them is $\{a, b\}$. Now, assume that c and d join group G at approximately the same time. Assume also that c is topologically close to a and that d is topologically close to b, as illustrated in figure 4. It is highly probable that a will receive CONGRESS' notification about c joining G before receiving notification about d joining G, and that b will receive the notifications in the reverse order (i.e., the notification about d will be received before the notification about c at b). This implies that a will calculate the membership of G first as $\{a, b\}$, then as $\{a, b, c\}$ and finally as $\{a, b, c, d\}$, whereas b will calculate the membership first as $\{a, b\}$, then as $\{a, b, d\}$ and only then as $\{a, b, c, d\}$.

Since the network is asynchronous and protocol messages may be delayed, membership information at distinct CONGRESS servers may differ at any given time. It has been proven that it is impossible to make strong guarantees regarding the preciseness of the membership derived from notifications at instable time periods (please see discussion in Section 5.1.1). If, however, the network stabilizes, and no new membership events occur in group G, then eventually all the members of G will have a *precise* view of the membership in G. This is formulated in the following property:

PROPERTY 4.2 (Eventual Preciseness). *Let G be a group, and t_0 a point in time s.t. the set of members of G has not changed after time t_0. Furthermore, assume that after time t_0 all the members of G are in the same connected component,* *and do not suspect each other. Then there is a time $t_1 > t_0$ s.t. at time t_1 all the members of G have the same computed membership, which consists of exactly the set of members of G.*

Consider, for instance, Example 4.1 above. There, the membership eventually becomes $\{a, b, c, d\}$ at all the processes.

Two points are worth noting about the above definition:

1. If there are live members of G in two disjoint network components, then we would like processes in each component to have a computed membership consisting of the set of members of G in the local component. It is easy to see that this requirement is fulfilled by any protocol that fulfills Property 4.2.
2. For simplicity's sake, we required in this paper that stability would last forever. In practice, however, the following situation holds: Let t_1 reflect a time by which all the events that occurred before time t_0 have been propagated to all the processes. If the membership of G stabilizes for only a finite time interval $[t_0, t_2]$, s.t. $t_2 > t_1$ then all the members of G will also have the same computed membership. This membership will consist of exactly the set of members of G during the interval $[t_1, t_2]$.

The stronger guarantees are formulated and proven in [**Ank97**].

5. Virtually synchronous membership services

Some applications require membership services with only weak semantics, and some require strong semantics. Applications that need to consistently maintain a replicated state (e.g., coherent cache), greatly benefit from virtually synchronous communication and membership semantics. The MOSHE membership service provides such semantics for groups that explicitly request this service. We call such groups *VS groups*. The protocol that implements these semantics is the MOSHE VS membership protocol.

There are many different formulations of group membership services (some examples may be found in [**VKCD98, DMS94, DMS95, BDM97**]), and various definitions of virtual synchrony semantics (e.g., [**VKCD98, BJ87, FvR95, MAMSA94**]). Our protocol provides semantics which have been proven useful for several distributed applications [**ABCD96, KD96, FLS97, ADMSM94, SM98**]. In Section 5.1 we specify the semantics provided by the MOSHE VS membership protocol.

Numerous group membership protocols providing similar semantics were constructed (e.g., [**CS95, AMMS$^+$95, MMSA$^+$96, EMS95, ADKM92, MPS91, MSMA91, DMS94, MS94, BDM97**]). The novelty of MOSHE is in its client-server approach: The servers maintain the membership of clients in groups. The client-server design is a major challenge, since the protocol has to synchronize different entities. Our implementation focuses on minimizing the number of messages sent in order to achieve preciseness, without sacrificing efficiency. In Section 5.2 we describe an overview of the implementation of the membership protocol.

5.1. Semantics of the MOSHE VS membership protocol. The membership protocol encapsulates membership notifications in views. A view v is a triple consisting of: the group name, denoted $v.G$; the group membership (i.e., the list of members), denoted $v.M$, and a view identifier, denoted $v.id$. We say that the view

v occurs in the group $v.G$. The view identifier is taken out of some totally ordered set.

The key features of the membership provided by MOSHE for VS groups are:

- Agreement on the order of views.
- Synchronization of multicast messages with view reports (*virtual synchrony*).

We elaborate on these features below.

5.1.1. *Agreement on views.* Agreement on the order of views allows processes that continue together to perceive changes of the membership in the same order. With the CONGRESS basic service, two processes that continue together may receive membership notifications in different order, as illustrated in Example 4.1 in Section 4.4. The membership service uses CONGRESS incremental membership notifications in conjunction with a one round agreement protocol, which allows processes that remain connected to receive views in the same order. The MOSHE VS membership protocol guarantees the following properties:

PROPERTY 5.1 (View Identifier Local Monotony). *Processes receive view reports in the view identifier order.*

PROPERTY 5.2 (Self Inclusion). *Processes deliver only views of which they are members.*

PROPERTY 5.3 (Agreement on Views). *Let G be a VS group, t_0 a point in time, and S a set of clients. Assume that from time t_0 onwards:*

1. *All the clients in S are members of G.*
2. *All the clients in S and their servers are in the same connected component C, and the topology of C does not change.*
3. *Processes in C do not suspect each other, and every process which is not in C is suspected by every member of C[2].*

Then there is a time $t_1 > t_0$ s.t. all the processes in S incur the same sequence of views in G after time t_1. Furthermore, if the set of members of G in C is exactly S after time t_0, then all the members of S eventually deliver the same view v, s.t. $v.M = S$ and do not deliver any further views in G.

As noted in Section 4.4, stability does not have to actually hold forever. It only has to hold "long enough" for the membership protocol to stabilize.

We would like to note that perfectly precise membership services are impossible to implement in truly asynchronous environments [**CHTCB96, VKCD98**]. A powerful adversary that fully controls the communication can force every deterministic membership algorithm to be imprecise, to block, or to constantly change its mind. Therefore, we have formulated Properties 4.2 (Eventual Preciseness) and 5.3 (Agreement on Views) to guarantee preciseness of the membership service only at stable periods in which the external failure detector module does not suspect correct and connected processes. If the failure detector is highly unreliable, then it is possible that our membership algorithm would never be precise. If, however, the failure detector is an *eventual perfect* one (please see [**CT96, DFKM96, DFKM97**]), and the communication stabilizes, then our membership service is guaranteed to eventually be precise.

[2]This requirement is fulfilled if the failure detector is eventually perfect. Please see [**CT96, DFKM96, DFKM97**].

5.1.2. *Virtual synchrony.* Virtual synchrony involves synchronizing multicast communication with membership notifications. In this programming model, group multicast send and receive events occur within the context of views. We say that a multicast send (receive) event e in group G occurs at process p in view v if v was the latest view that p received in group G before e. The MOSHE VS membership protocol guarantees:

PROPERTY 5.4 (Self Delivery). *A message sent by a process is eventually delivered by that process, unless the process suffers a crash failure or leaves the group.*

PROPERTY 5.5 (Termination of Delivery). *If a process p sends a message in a view v in G, and process q is in $v.M$, then either q delivers this message or p eventually delivers a new view in G.*

PROPERTY 5.6 (Synchronous Delivery). *Every message is delivered within the view in which it was sent*[3].

In addition, MOSHE provides the following property for groups that require it:

PROPERTY 5.7 (View Synchrony). *Two processes undergoing the same two consecutive views in a group G deliver the same set of messages in G within the former view.*

The MOSHE VS guarantees are very similar to the *Extended Virtual Synchrony* semantic described in [**MAMSA94**]. We have removed the causal order, total order, and safe delivery properties from that semantic, in the belief that these are optional properties that one might build on top of this service. There are other group membership specifications in the literature, such as [**FvR95**], [**BDM97**] and [**DMS95, VKCD98**]. These differ in various details, but have much in common with the semantic that is used here.

5.2. Implementation of the MOSHE VS membership protocol. In this section we describe an overview of the implementation of the MOSHE VS membership protocol. The implementation details may be found in [**KSDM**].

The MOSHE membership algorithm maintains the list of members in each group. The algorithm is invoked in response to requests from clients to join or leave groups, and in response to network events. MOSHE servers extend CONGRESS LMSs. MOSHE uses CONGRESS incremental updates to propagate knowledge of network changes and of membership events.

5.2.1. *Message flow in* MOSHE*.* When a client join/leave request or a client failure report is handled by a MOSHE server, the server invokes a corresponding join/leave request in CONGRESS for further dissemination among the other membership servers. Once CONGRESS generates a membership notification reflecting this event, MOSHE starts a synchronization round, in order to achieve virtual synchrony and agreement on the view.

The membership server computes a proposed view, as explained in Section 5.2.3 below, and emits *proposals* reflecting it to all of its clients which are members of the group. The clients echo the proposals in *flush* messages, which acknowledge their participation in this view. The flush messages are propagated to all the membership servers. The clients also multicast the flush messages to the other members of the

[3] This can be relaxed in various ways, which are not in the scope of this paper. Please see [**FvR95, SM98, VKCD98**].

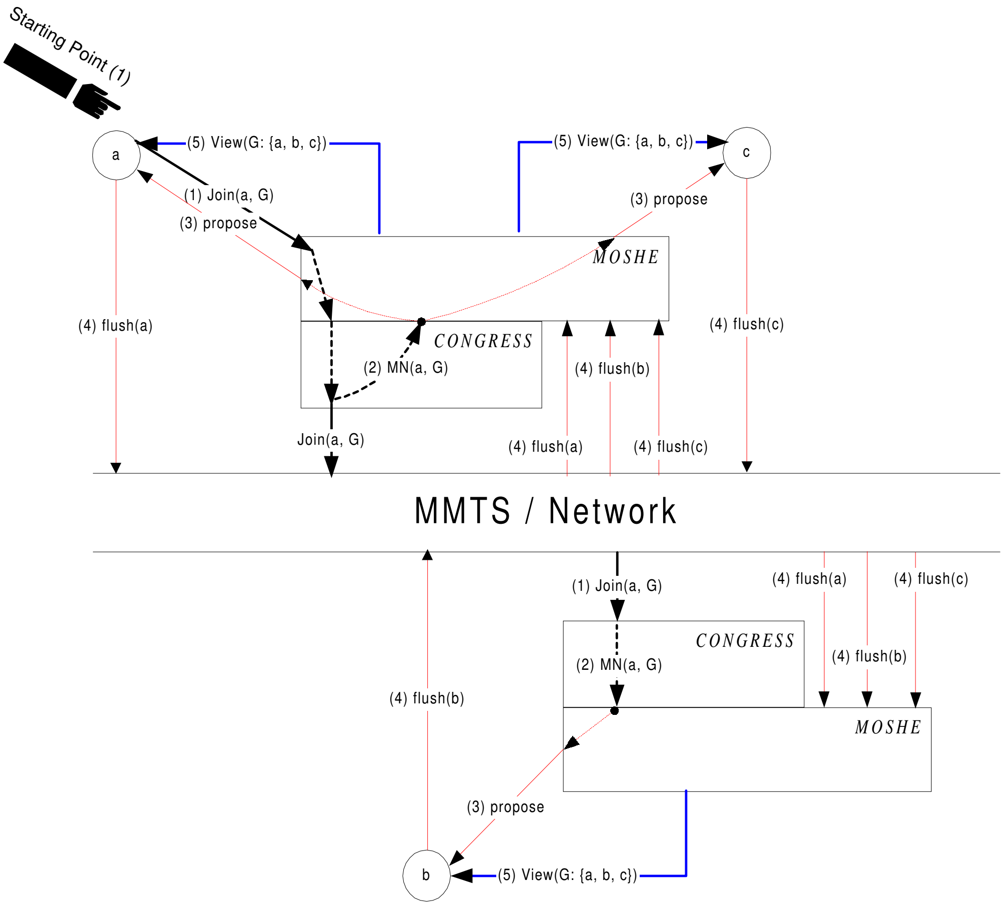

FIGURE 5. Events occurring in MOSHE when process a is joining VS group G.

group in order to synchronize view delivery with the multicast message flow, as explained in Section 5.2.2 below.

After the proposals are emitted, we say that the view is *pending* until flush messages arrive from all of its members. If a new incremental membership notification arrives while the view is pending, then new proposals are sent only to those newly joined clients to whom proposals for this view were not previously sent.

Once the MOSHE server receives flush messages from all the members of the pending view, it emits a view message to all of its clients (which are members of the group). The clients synchronize the view with the flush messages, as explained in Section 5.2.2 below.

EXAMPLE 5.1. In Figure 5 we illustrate the events that occur in the protocol when process a is joining group G. Initially, the membership of group G is $\{b, c\}$. The protocol is invoked when process a issues its join request.

The join request (denoted by (1) Join(a, G) in Figure 5), is propagated using CONGRESS. All the MOSHE servers learn of it via the CONGRESS membership notification, denoted by (2) MN(a, G).

Upon receiving this event, the MOSHE servers generate proposals for a new membership to their clients. In Figure 5, these are denoted by (3) propose. The clients respond by multicasting (4) flush messages to the servers and clients.

Once the MOSHE membership server receives flush messages from all the members of the new view, it issues a view message, (denoted by (5) View(G: $\{a, b, c\}$)).

5.2.2. *Supporting virtual synchrony.* Virtual synchrony requires synchronization among the clients: In order to fulfill Properties 5.6 (Synchronous Delivery) and 5.7 (View Synchrony) the clients have to synchronize their multicast messages with membership events. Such synchronization necessarily incurs a delay period in which messages may not be transmitted [**FvR95**].

The synchronization mechanism is based on the flush messages described above. The purpose of flush messages is to synchronize views with the multicast message flow. Therefore, flush messages are multicast via the MMTS, and serve as *place holders* which denote where (in the message flow) the previous view ends. After sending a flush message, the clients do not send any new messages until the new view is reported.

Once the MOSHE membership server receives flush messages from all the members of the new view, it issues a view message. The view also contains a view identifier as explained in Section 5.2.3 below. The clients synchronize the view with the flush messages: Messages that were sent before the flush are delivered in the previous view. Recall that we assume that the MMTS provides FIFO multicast services; hence, messages sent before the flush message are delivered before the flush message. This way, every message is delivered in the view in which it was sent.

5.2.3. *Computing the proposed view.* A proposed view (for a group G) consists of the proposed set of members, a proposed view identifier, and a proposed set of suspects (or leavers).

The membership server computes the initial members and suspects sets of the proposed view by applying the CONGRESS incremental membership notification to the membership of the current view. When a CONGRESS incremental membership notification reports that a process is leaving a group[4], the process is removed from the members set and added to the suspects set. In case a join notification arrives, the joiner is added to the members set and the suspects set is empty.

Consider Example 5.1 above: There, the membership of group G was $\{b, c\}$ and the incremental notification reported that a had joined. The proposed set of members is therefore $\{a, b, c\}$, and the proposed set of suspects is empty.

The proposed view identifier is computed by incrementing the latest known view identifier by one. For example, if the latest view was $< G, \{b, c\}, 3 >$, then the new proposed view is $< \{a, b, c\}, \{\}, 4 >$. The proposed view is sent in the proposal, and echoed in the corresponding flush message.

If a new incremental notification arrives while the view is pending, then the pending view is re-computed by aggregating the incremental notification to the pending view. When a join notification arrives, the joiner is added to the members

[4]This can occur either because the process crashed or because it has requested to leave the group.

set unless it is already in the suspects set[5]. New proposals are then sent only to those newly joined clients to whom proposals were not previously sent. This is illustrated by the following example:

EXAMPLE 5.2. Consider Example 4.1 (illustrated in Figure 4) where two processes, c and d, concurrently try to join group G. Assume that the current view of group G is $< G, \{a, b\}, 3 >$. Consider the case in which membership server M_1 first receives the notification that c is joining, and then issues proposals for the view $< G, \{a, b, c\}, \{\}, 4 >$ to a and c. At the same time, server M_2 receives the notification that d is joining, and then issues proposals for the view $< G, \{a, b, d\}, \{\}, 4 >$ to b and d.

In the next stage, M_1 receives the notification that d is joining, and aggregates it to the pending view, which now becomes $< G, \{a, b, c, d\}, \{\}, 4 >$. Server M_1 checks if it has to send new proposals: Since the only new member (d) is not a client of M_1, no new proposals have to be emitted. Similarly, the aggregated pending view at server M_2 becomes $< G, \{a, b, c, d\}, \{\}, 4 >$, and M_2 does not emit new proposals.

Similarly, if a leave notification is received during the synchronization round, the server removes the leaving client from the members set of the pending view, adds it to the suspects set, and no longer waits for a flush message from this client. In order to prevent blocking, the servers should eventually either receive a flush message from every member of the view, or receive a notification that the member has failed. This is fulfilled since we assume that the failure detector is complete, i.e., eventually suspects every faulty process.

Note that it is possible for a flush message that reflects a CONGRESS membership notification to arrive before (or even without) the membership notification[6]. In such cases, the incremental change reflected in the flush message is also aggregated into the pending view: The suspects set becomes the union of the suspect sets, and the new members set consists of the members of the union of the members sets, except for those who are in the suspects set.

Once flush messages arrive from all the members of the pending view, the server sends the new view to the clients. The view membership is the members set of the pending view, and the view identifier is chosen to be the maximum of the proposed view identifiers among the collected flush messages.

Consider Example 5.2 above: There, the servers wait until they receive flush messages from all four members before they send the new view to their clients. Since all the proposals contain the proposed view identifier 4, this is the view identifier for the new view. Thus, two separate CONGRESS incremental membership notifications are aggregated and reflected in one view: $< G, \{a, b, c, d\}, 4 >$.

In case two previously disconnected servers become connected, it is possible that they may send proposals for the same view with different view identifiers. This is illustrated in the example below.

EXAMPLE 5.3. There are two membership servers, M_1 and M_2, which are disconnected due to a network failure. At M_1 the view of group G is $< \{a, b\}, 3 >$, while at M_2 the view is $< \{c, d\}, 5 >$. There is a difference in the view identifiers

[5] If the joiner is in the suspects set, the notification is buffered to be handled after the current pending view will be delivered.

[6] This can occur, for example, in case of a false suspicion at some of the processes.

due to a couple of membership changes that occurred at M_2's network component while M_1 and M_2 were disconnected.

Now, the network failure is mended and all the processes in the system reconnect. M_1 receives a CONGRESS membership notification that reflects the join of c and d, and M_2 receives a CONGRESS membership notification that reflects the join of a and b. M_1 emits the proposal $< G, \{a, b, c, d\}, \{\}, 4 >$ whereas M_2 emits the proposal $< G, \{a, b, c, d\}, \{\}, 6 >$. These proposed views are echoed in the flush messages. Once all the flush messages are collected, both servers report of the view $< G, \{a, b, c, d\}, 6 >$, since 6 is the maximum view identifier among the collected flush messages.

5.2.4. *Recovery from server failures.* The MOSHE service is fault tolerant: If a MOSHE server crashes, its clients are transparently migrated to another MOSHE server, and the application program is unaware of this change.

When a client receives a server failure report, it tries to reconnect to an alternative server[7]. The live servers also receive a report of the server's failure via CONGRESS. When a live MOSHE server receives such a report, it waits for the failed server's clients to connect to it during a predefined time interval, called the *reconnection time interval.*

After the reconnection time interval is over, the servers exchange among themselves the list of reconnected clients, and issue a leave event for those clients that did not succeed to reconnect. Since the system is asynchronous, it is possible that a client will succeed to reconnect to a server only after the reconnection time interval is over. In this case, when the client reconnects the server notifies it that the reconnect is too late. The client then re-joins all the groups that it was previously a member of.

During the migration period, some messages that were in transit between the client and server may have been lost. We now describe how the protocol recovers from these message losses:

Recovery from a lost proposal or flush message: After the reconnection time interval is over, each server emits proposals for pending views to the reconnected clients. The clients echo these proposals in flush messages as usual. The servers ignore duplicate flush messages, and ignore flush messages which pertain to views that have already been cleared.

Recovery from a lost view report: When a client connects to a new server it emits a join request for each group that corresponds to a pending view (a pending view is one for which the client received a proposal and has not received a view report yet). If the server receives a join request from a client that is included in the group membership, it assumes that the client had lost the view report, and re-sends it.

Recovery from a lost join or leave: The client re-issues join/leave requests for every group that it tried to join/leave but did not receive a proposal reflecting this request. In order to keep track of these groups, the client also needs to receive proposals for groups that it is leaving. Such proposals are not echoed in flush messages.

[7]The alternative server may be located using CORBA services, or using a list of alternative servers that the client holds. For details please see [**Now98**].

6. Advanced group membership services

The client-server design of the membership service allows us to support a variety of advanced services without adding complexity to the clients and hence without paying a performance penalty. MOSHE provides advanced services such as hierarchical organization of groups, secure groups and group policies.

An important innovation of our membership service is the support for hierarchical directory services. MOSHE maintains a hierarchy of groups: a group may be a *sub-group* of a parent group. A parent group may contain a number of sub-groups. This concept is useful for applications containing a number of logically related groups, e.g., a conferencing application with several discussion groups.

MOSHE also supports secure group services: It implements authentication and authorization mechanisms that determine when a user is authorized to perform actions in a group, (e.g., create a sub-group for a specified group, query which sub-groups a group has, or join a group). Furthermore, the membership service may maintain two membership sets for each process group: *active members* who may provide input in the group, and *passive members* who receive messages sent to the group but cannot send messages to the group.

Thus, the authentication mechanism allows users to determine *policies* which restrict the ability of processes to become (active/passive) members of the group. The policies are declared when the group is created. If no policy is declared, then the policies are inherited from the parent group.

More sophisticated policies may be also imposed, e.g., restricting the number of members in a group, or even the properties of the members. For example, a cosmopolitan conference over the Internet may allow only two members from each country to participate in the discussion. If, due to a membership policy, a user's join request may not be currently fulfilled but may possibly be fulfilled later, MOSHE allows the user to *block* until the join will become possible.

7. Conclusions

In this paper we presented a novel membership service, which exploits the client-server paradigm to allow "thin" lightweight clients to benefit from sophisticated membership services.

Our membership service provides two different service levels and semantics, geared to different applications with different needs: the CONGRESS membership service which provides simple semantics of membership approximation, and the MOSHE service which extends it to provide full virtual synchrony semantics for applications that require consistency.

CONGRESS minimizes the communication overhead and is therefore scalable and appropriate for WANs. CONGRESS uses a logical name space for group addressing and enables maintenance of dynamic multicast groups. The protocol copes with network and host failures, and exploits the network's hierarchical addressing scheme to support world-wide scalability and scoping.

The MOSHE VS membership protocol provides agreement on membership and strong semantics of message ordering w.r.t. membership changes, namely, virtual synchrony [**BJ87, FvR95, MAMSA94, VKCD98**]. Virtual synchrony requires synchronization among the applications and the membership service. This synchronization greatly facilitates the design of applications that require consistency (e.g., applications with shared data [**BJ87, ABCD96, KD96, FLS97, ADMSM94**]), but is too costly for applications that require real-time message delivery (e.g., video transmission). Therefore, we provide virtually synchronous communication only for groups that explicitly request this service.

We have implemented MOSHE in Java. Clients may communicate with the MOSHE server using the standard CORBA interface, or using a TCP/IP socket interface. MOSHE also provides some advanced services such as a hierarchical directory of groups and secure group membership services. MOSHE is inherently fault tolerant: In case of a server failure, its clients are transparently migrated to another server.

Acknowledgements

We are grateful to Aviva Dayan for many helpful comments which greatly helped improve the quality of the presentation.

Many people contributed to the design and implementation of the membership services. David Breitgand and Zohar Levy participated in the design of CONGRESS. Jeremy Sussman participated in the design and implementation of the VS membership algorithm. Gabriel Benhanokh and Ariel Nowersztern participated in the implementation of the communication framework for MOSHE; Ariel Nowersztern implemented the recovery from server failures, and Gabriel Benhanokh implemented the authorization and authentication mechanisms in MOSHE. Gad Admoni implemented the hierarchical directory of groups.

References

[ABCD96] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev, *Group communication as an infrastructure for distributed system management*, 3rd International Workshop on Services in Distributed and Networked Environment (SDNE), June 1996, pp. 84–91.

[ABDL96] T. Anker, D. Breitgand, D. Dolev, and Z. Levy, CONGRESS*: CONnection-oriented Group-address RESolution Service*, Tech. Report CS96-23, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, December 1996, Available from: `http://www.cs.huji.ac.il/~transis/`.

[ABDL97] T. Anker, D. Breitgand, D. Dolev, and Z. Levy, CONGRESS*: Connection-oriented group-address resolution service*, Proceedings of SPIE on Broadband Networking Technologies, November 2-3 1997.

[ACK+97] T. Anker, G. Chockler, I. Keidar, M. Rozman, and J. Wexler, *Exploiting group communication for highly available video-on-demand services*, Proceedings of the IEEE 13th International Conference on Advanced Science and Technology (ICAST 97) and the 2nd International Conference on Multimedia Information Systems (ICMIS 97), April 1997, pp. 265–270.

[ACM96] ACM, *Commun. acm 39(4), special issue on Group Communications Systems*, April 1996.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki, *Membership algorithms for multicast communication groups*, 6th International Workshop on Distributed Algorithms (WDAG), November 1992, pp. 292–312.

[ADMSM94] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser, *Robust and Efficient Replication using Group Communication.*, Tech. Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[AMMS+93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, *Fast message ordering and membership using a logical token-passing ring*, 13th International Conference on Distributed Computing Systems (ICDCS), May 1993, pp. 551–560.

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, *The totem single-ring ordering and membership protocol*, ACM Trans. Comput. Syst. **13** (1995), no. 4.

[Ank97] T. Anker, CONGRESS: *CONNection-oriented Group-address RESolution Service*, Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1997.

[BDM97] Ö. Babaoğlu, R. Davoli, and A. Montresor, *Partitionalbe Group Membership: Specification and Algorithms*, TR UBLCS97-1, Department of Conmputer Science, University of Bologna, January 1997.

[BFHR98] K. Birman, R. Friedman, M. Hayden, and I. Rhee, *Middleware support for distributed multimedia and collaborative computing*, Multimedia Computing and Networking (MMCN98), 1998, To appear.

[Bir96] K. Birman, *Building Secure and Reliable Network Applications*, ch. 18, Manning, 1996.

[BJ87] K. Birman and T. Joseph, *Exploiting virtual synchrony in distributed systems*, 11th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), ACM, Nov 1987, pp. 123–138.

[Car94] Georg Carle, *Reliable group communication in ATM networks*, Proceedings of the Twelve Annual Conference on European Fibre Optic Communications and Networks EFOC&N'94, June 21–24 1994.

[CHKD96] G. Chockler, N. Huleihel, I. Keidar, and D. Dolev, *Multimedia multicast transport service for groupware*, TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies, September 1996, Full version available as Technical Report CS96-3, The Hebrew University, Jerusalem, Israel.

[CHTCB96] T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, *On the impossibility of group membership*, ACM Symposium on Principles of Distributed Computing (PODC), May 1996, pp. 322–330.

[CS95] F. Cristian and F. Schmuck, *Agreeing on Process Group Membership in Asynchronous Distributed Systems*, Tech. Report CSE95-428, Department of Conmputer Science and Engineering, University of California, San Diego, 1995.

[CT96] T. D. Chandra and S. Toueg, *Unreliable failure detectors for reliable distributed systems*, J. ACM **43** (1996), no. 2, 225–267.

[DFKM96] D. Dolev, R. Friedman, I. Keidar, and D. Malki, *Failure Detectors in Omission Failure Environments*, TR 96-13, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, September 1996, Also Technical Report 96-1608, Department of Computer Science, Cornell University.

[DFKM97] D. Dolev, R. Friedman, I. Keidar, and D. Malki, *Failure detectors in omission failure environments*, ACM Symposium on Principles of Distributed Computing (PODC), August 1997, Brief announcement.

[DMS94] D. Dolev, D. Malki, and H. R. Strong, *An Asynchronous Membership Protocol that Tolerates Partitions*, Tech. Report CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[DMS95] D. Dolev, D. Malki, and H. R. Strong, *A Framework for Partitionable Membership Service*, TR 95-4, Institute of Computer Science, The Hebrew University of Jerusalem, March 1995.

[EMS95] P. D. Ezhilchelvan, A. Macedo, and S. K. Shrivastava, *Newtop: a fault tolerant group communication protocol*, 15th International Conference on Distributed Computing Systems (ICDCS), June 1995.

[FJM+95] Sally Floyd, Van Jacobson, Steven McCanne, Ching-Gung Liu, and Lixia Zhang, *A reliable multicast framework for light-weight sessions and application level framing*, Proceedings of the IEEE/ACM Transactions on Networking., November 1995, An earlier version of this paper appeared in ACM SIGCOMM 95, August 1995, pp. 342–356.

[FLS97] A. Fekete, N. Lynch, and A. Shvartsman, *Specifying and using a partionable group communication service*, 16th ACM Symposium on Principles of Distributed Computing (PODC), August 1997.

[FvR95] Roy Friedman and Robbert van Renesse, *Strong and Weak Virtual Synchrony in Horus*, TR 95-1537, dept. of Computer Science, Cornell University, August 1995.

[KD96] I. Keidar and D. Dolev, *Efficient message ordering in dynamic networks*, 15th ACM Symposium on Principles of Distributed Computing (PODC), May 1996, pp. 68–76.

[KSDM] I. Keidar, J. Sussman, D. Dolev, and K. Marzullo, *A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership*, In preparation.

[MAMSA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, *Extended virtual synchrony*, 14th International Conference on Distributed Computing Systems (ICDCS), June 1994.

[MMSA+96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, *Totem: A fault-tolerant multicast group communication system*, Commun. ACM **39** (1996), no. 4.

[MPS91] S. Mishra, L. L. Peterson, and R. D. Schlichting, *A Membership Protocol based on Partial Order*, Proc. of the intl. working conf. on Dependable Computing for Critical Applications, Feb 1991.

[MS94] C. Malloth and A. Schiper, *View synchronous communication in large scale networks*, 2nd Open Workshop of the ESPRIT project BROADCAST (Number 6360), July 1995 (also available as a Technical Report Nr. 94/84 at Ecole Polytechnique Fédérale de Lausanne (Switzerland), October 1994).

[MSMA91] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala, *Membership algorithms for asynchronous distributed systems*, International Conference on Distributed Computing Systems (ICDCS), May 1991.

[Now98] A. Nowersztern, *MOSHE: Membership Object-oriented Service for Heterogeneous Environments*, Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, January 1998, Available from: `http://www.cs.huji.ac.il/labs/transis/`.

[PSK94] Sanjoy Paul, Krishan K. Sabnani, and David M. Kristol, *Multicast transport protocols for high speed networks*, Proceedings of the International Conference on Network Protocols, 1994, pp. 4–14.

[PSLB97] Sanjoy Paul, K. Sabnani, J.C. Lin, and S. Bhattacharyya, *Reliable multicast transport protocol (RMTP)*, IEEE Journal on Selected Areas in Communications (1997).

[Rod91] Tom Rodden, *A survey of CSCW systems*, Interacting with Computers **3** (1991), no. 3, 319–353.

[SM98] J. Sussman and K. Marzullo, *The* bancomat *problem: An example of resource allocation in a partitionable asynchronous system*, 12th International Symposium on DIStributed Computing (DISC), September 1998, To appear.

[VKCD98] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev, *Group Communication System Specifications: A Comprehensive Study*, Tech. report, Institute of Computer Science, The Hebrew University of Jerusalem, 1998, In preparation.

[Vog96] Werner Vogels, *World wide failures*, Proceedings of the ACM SIGOPS 1996 European Workshop, September 1996.

Institute of Computer Science, The Hebrew University of Jerusalem, Givat Ram, Jerusalem 91904, Israel

E-mail address: `anker@cs.huji.ac.il`

Institute of Computer Science, The Hebrew University of Jerusalem, Givat Ram, Jerusalem 91904, Israel

E-mail address: `grishac@cs.huji.ac.il`

Institute of Computer Science, The Hebrew University of Jerusalem, Givat Ram, Jerusalem 91904, Israel

E-mail address: `dolev@cs.huji.ac.il`

Institute of Computer Science, The Hebrew University of Jerusalem, Givat Ram, Jerusalem 91904, Israel

E-mail address: `idish@cs.huji.ac.il`