Group Communication

1 Papers

- Flaviu Cristian and Frank Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, University of California-San Diego, La Jolla, CA 92093-0114, 1995.
- Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. ACM Computing Surveys, 33(4):1–43, December 2001.
- I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. In 20th International Conference on Distributed Computing Systems (ICDCS), pages 356–365, Taipei, Taiwan, April 2000. Full version: MIT Technical Memorandum MIT-LCS-TM-593.

2 Introduction

Group communication lets group of processes communicate. A group is e.g. participants in a videoconference, replicas for a data item.

Participants in the group may change over time, as processes join, leave or fail. *View-oriented group communication systems (GCS)* is a group communication system for groups with dynamic membership. A GCS consists of two services, *membership* and *multicast*.

Membership service of GCS keeps track of who's in the group.

Multicast service allows processes to send messages to the group with some *quality of service (QoS)* guarantees. *QoS* refers to ordering and reliability of messages.

Membership and multicast have separate specs. Also, each problem is interesting in its own right.

This lecture covers specs for GCSs in general. It also presents two group membership algorithms, one for LANs, a second for WANs.

3 GCS Specifications

From the Chockler, Keidar, Vitenberg paper.

We specify a number of useful properties that GCSs should satisfy.

3.1 Model

Typically we consider an asynchronous message passing network, with lossy point-to-point links between all processes.

Allow process crash and recovery, and link failure and recovery (partitions and merges).

A *view* is a set of processes belonging to a group. Views are labeled (but the same set of processes may be associated with different labeled views).

General model is a GCS service, receiving $crash/recovery$ events from the environment. The GCS service receives *send*s from application, and outputs *recv*, and *view_chg*s to the application. The signature is as follows:

- env input $crash(p)$: process p crashes.
- env input $recover(p)$: p recovers.
- app input $send(p, m)$: p sends message m.
- output to app $recv(p, m)$: p receives m.
- output to app $view\_chg(p, \langle id, members \rangle, T)$: p receives message that new view is id, consisting of processes in $members$. T is the *transition set* (later).

General model may be modified for particular algorithms, as we'll see for the two membership algorithms.

Below, we list a number of useful properties that GCSs should satisfy. We'll also try to give examples of why these properties are useful. However, we won't discuss how to achieve (i.e. implement a GCS with) these properties. We'll see some implementations in the two membership algorithms later today, and in the following lecture. We'll first discuss safety properties of the membership service, then safety properties of the multicast service, the ordering and reliablity properties of the multicast service, and finally some liveness properties.

3.2 Membership Service: Safety Properties

Assume for simplicity that all live processes belong to a single group. Also assume that no processes leave or join the system. Thus, group membership changes only when processes and links crash or recover.

In the *ideal* membership service, each process has a view of the members of its group. Each time a process fails / recovers or a link goes up / down, all the remaining members of the group instantly know the new membership. Of course, it's impossible to implement this service in an unreliable, asynchronous network. In general, the members of a group won't know instantly when their membership changes, and they also won't always agree on what the exact membership is. Instead, what we implement is a *membership service (MS)* which monitors the status of all the processes and links, acting like a failure and recovery detector. The MS keeps members of a group approximately up-to-date on their membership by sending *view_chg* events to the group each time it detects some process / link in the group fails / recovers. Each view the MS sends is labeled by a value from a totally ordered set, and views are ordered by their labels. Say a process *installs* a view when it receives a *view_chg* with that view.

We want the MS to satisfy the following basic properties:

- *Self Inclusion*: A process is a member of any view it installs.
- *Local Monotonicity*: The views that a process installs increase in time.
- *Initial View Event*: Every event (e.g. *send/recv/view_chg*) at a process occurs in some view.

Self inclusion is clearly necessary, since a process can't do anything in a group it doesn't belong to. Initial view event is useful for the technical reason that all events are supposed to occur in some view. Local monotonicity is useful because it ensures that if two processes both install the same two views, they install them in the same order.

There is a question of how many "active" groups we want to allow concurrently. The right answer depends on the specific application. A MS that allows multiple active groups is called *partitionable*. If it only allows one active group, it is called *primary component*. A primary component MS satisfies the following property:

- *Primary Component Membership*: The set of views installed by all the processes can be totally ordered. For any two consecutive views in the ordering, there must be a process in the smaller view which installed the larger view.

To see this property is useful, consider a consistent data replication algorithm using a primary component MS. Since there is some process which survives between any two consecutive views, that process can be used to transfer the value of the data consistently between the views. Without this property, there could be two concurrent views with no process in common. If different updates were applied in the two views, the data would have inconsistent values.

3.3 Multicast Service: Safety Properties

Two basic properties the multicast service should satisfy are:

- *Delivery Integrity*: For every receive event there is a preceding send event of the same message.
- *No Duplication*: No message is received twice.

Both these properties say there are no spurious receive events. For the no duplication property, we assume that each message sent is unique (e.g. tagged by a sequence number).

A *multicast service (MCS)* is only useful if the messages a process sends and receives using it has some correspondence with the view(s) that the process belongs to over time. This is expressed in the next two properties.

- *Sending View Delivery*: If a process receives a message in some view, then the message was sent in that view.
- *Same View Delivery*: All processes which receive some message receive it in the same view.

Sending view delivery is useful for minimizing the amount of *context information* that must be sent with each message, and the amount of processing per message. It is also useful for applications that only care about a message that is received in the same view it was sent, e.g. messages for *state transfer* when changing views. However, to satisfy sending view delivery without discarding messages from live processes, the GCS must sometimes *block* view changes while messages from the old view are delivered. To avoid this situation, same view delivery would suffice for applications which don't care what view a message was sent in.

While sending view delivery allows us to discard out of date state transfer messages, it doesn't tell us when we need to send them. The next three properties deal with this.

- *Virtual Synchrony*: If two processes in a view V install the same new view, then the processes receive the same messages in V.
- *Transitional Set*: If processes p and q install the same view, then q is included in p's *transitional set* iff p and q had the same previous view.
- *Agreement on Successors*: If a process p installs view V in view V', then any process q which installs V and is a member of V' also installs V in V'.

Virtual synchrony is perhaps the best known property of GCSs. It lets an application avoid doing *state transfers* for a view change in some cases. In fact, consider two processes in a view which install the same new view. Then since these two processes received the same messages in the previous view, there is no need to do state transfer between them in the new view. But it's not enough for a GCS to provide virtual synchrony, it must also be able to tell processes in a view when they have actually continued together into a new view. Processes can obtain this information if the GCS guarantees either the transitional set property, or agreement on successors, in addition to virtual synchrony.

3.4 Multicast Service: Ordering and Reliability Properties

We studied various kinds of quality of service (QoS) for broadcasts and multicasts in lecture 10. Those ideas can be extended to group communication as well. Recall that ordering properties specify the order that messages are received in relation to the order in which they were sent, and in relation to the order which other processes receive them. Reliability properties specify that messages which are sent must be received in some cases.

The basic ordering and reliability guarantees are:

- *FIFO delivery*: If a process sends two messages, then every process that receives both messages receives them in the order they were sent.
- *Reliable FIFO*: If a process sends two messages, then any process that receives the later message receives the earlier message first.
- *Causal Delivery*: If message m causally precedes message m', then every process that receives both messages receives m before m'.
- *Reliable Causal*: If message m causally precdes m' and both are sent in the same view, then any process that receives m' receives m earlier.

There are three types of totally ordered broadcasts:

- *Strong Total Order*: There is a total ordering f on the messages such that all the processes receive messages in an order consistent with f.

- *Weak Total Order*: For every pair of views V and V' there is a total order f on the messages so that every process that installs V in V' receives messages in V' in an order consistent with f.
- *Reliable Total Order*: There is a total order f on the messages such that if m and m' are two messages sent in the same view and m precedes m' in f, then any process that receives m' receives m earlier.

These properties are useful for implementing a replicated state machine.

We can also define message ordering constraints between messages with different QoS. In addition, if there are multiple groups, then we can define constraints on the order in which messages are received in different groups.

3.5 Liveness

Many of the properties listed above are similar to solving agreement problems between the processes. Thus, we know that there is no algorithm which achieves the properties in an asynchronous network which is guaranteed to terminate. But we can make some liveness guarantees about membership and multicast algorithms contingent on perfect failure detectors and stability in the network. We say that a set of processes forms a *stable component* if they are eventually alive and connected to each other, and the link from any process in this set to any process outside the set is down. We say that a failure detector is *eventually perfect* if it eventually always reports to the processes in any stable component S that their reachable set is S. We would like the GCS to provide accurate group membership information and successfully multicast messages when the network is stable and the GCS has access to an eventually perfect failure detector:

- *Liveness* If the GCS has access to an eventually perfect failure detector, then for every stable component S, there exists a view V with members set S such that for every process p in S, we have
  1. *Membership Precision*: p installs V as its last view.
  2. *Multicast Liveness*: Every message p sends in V is received by every process in S.

Having listed many properties we would like our membership and multicast services to have, we now proceed to describe two algorithms for implementing a membership service. The first algorithm is designed for a LAN environment, the second for a WAN.

4 Group Membership in Timed Asynchronous Systems

From the Cristian, Schmuck paper.

We consider three algorithms (we combine the two 3 round algorithms in the paper, and skip the 5 round algorithm) for implementing a group membership service in a LAN (paper doesn't say this but it's implicit). The algorithms successively improve on the consistency of the view they offer, and their efficiency. We will define consistency and efficiency as we go along.

4.1 Model

Processors connected by lossy, point-to-point asynchronous network.

System is *timed*. All processes have timers running at the *same rate*, and can take steps when timers go off.

Processors can suffer *crash/performance* failures, and recoveries. Performance failure is the processor or its clock running too fast or slow.

Network has *omission/performance* failures. Omission failure is message loss. Performance failure is untimely message delivery, i.e. message takes longer than time δ to send.

During a time interval, a link can be *connected, disconnected* or *partially connected*. The link is connected (disconnected) is if all (no) messages are delivered. It is partially connected is if only some messages are delivered.

A *partition* is a set of mutually connected processes (at some point in time).

The system is *stable* from some point on in time if no processes fail or recover, and no changes in network connectivity occur after that time.

4.2 Detecting System Changes

The goal of the membership service is to give each process a view of what other processes it is connected to, and to have some consistency between these views. To this end, each process monitors which processes it is connected to. When the connectivity changes, this may cause processes that notice the change to start running a membership algorithm to form new views.

4.2.1 Detecting Failures

Failures are detected, as always, by sending "alive" messages. However, to decrease the number of messages sent, the messages are sent in a *virtual ring*. That is, suppose there is a partition S, and all the processes in S know they are connected. Then the processes in S form themselves into a ring according to some rule (e.g. the order of their IDs), and some leader process in S (e.g. the one with the lowest ID) sends a "I am alive" message around the ring every π time units. All the processes expect to get the message every π time units. If they don't, they know some process in S has failed or disconnected, and they can start the membership algorithm.

4.2.2 Detecting Recoveries

Detecting recoveries is done by having live processes send probes. The reception of a probe indicates the sender is alive. However, again to decrease the number of messages sent, only the leader of a partition sends probe messages, every μ time units, and these messages list the members of S.

Lastly, note that both the failure and recovery detection mechanisms rely on the membership protocol, to determine the set of connected processes.

Each of the following three subsections lists a number of properties we want from the membership service, followed by an algorithm which achieves those properties. Compared to the membership service safety properties listed in Section 3.2, these properties are often more detailed and involved.

4.3 Minimal Properties

We assume each process p has a data structure $members(p)$, which stores the processes that are connected to p. We expect the membership to satisfy the following properties:

- *(Tjd) Bounded Join Delays*: There is a constant JD such that if the system is stable at time t, and two processes p and q are connected to each other, then $p \in members(q)$ and $q \in members(p)$ by time $t + JD$.
- *(Tpd) Bounded Partition Detection Delays*: There is a constant PD such that if the system is stable at time t, and two processes p and q are alive but disconnected from each other, then $q \notin members(p)$ and $p \notin members(q)$.
- *(Sld) Stability of Local Views*: If the system is stable at time t, then for any process p, $members(p)$ does not change after $t + \max(JD, PD)$.

Property Tjd says that if two processes are connected, they join the same group after a constant amount of time. Tpd says that if two processes are disconnected, they are not in the same group after a constant amount of time. Sld says that if the system is stable, then every process will eventually settle on a group.

4.3.1 Independent Assessment Protocol

The independent assessment protocol achieves the requirements of Section 4.3. This protocol is the only one in the paper which does *not* use the mechanisms in 4.2 for fault detection. Instead, every process just probes the other processes, and decides independently which processes it is connected to. More precisely, every process p initializes $members(p) = \{p\}$, then sends an "alive" message to every other processor every π time units. If process q receives an alive message from p, it adds p to $members(q)$. If it doesn't receive a message from p for $\pi + \delta$ time, it removes p from $members(q)$.

It is easy to see that the independent assessment protocol satisfies properties Tjd,Tpd and Sld. Also, we have $JD = \pi + \delta$, $PD = \pi + 2\delta$.

The problem with this protocol is that process p may think q is alive, while q doesn't think p is alive. This is called "split brain behavior".

4.4 Minimal Group Membership Service Properties

In the independent assessment protocol, every process just keeps a list of processes it believes it is connected to. What it lacked was the notion of a *group*. Note that "group" in the CS paper is the same as "view" in the CKV and KSMD paper. We will use group and view synonymously throughout Section 3. A group is a set of processes, and each group is labeled by a unique identifier from a totally ordered set. Two different groups can refer to the same set of processes. To improve the independent assessment protocol, we let processes join and leave groups. For each process p, $group(p)$ is the group (identifier) that p is joined to. Note that this definition depends on the time we are considering. However, for simplicity we will usually not explicitly mention the time. We let $members(p)$ be the set of processes in the group that p is joined to (at some point in time). We let $joined(p)$ be a predicate indicating whether p is joined to any group (at some point in time). Ideally, if S is a partition, then $members(p) = S$ for all $p \in S$. However, as noted earlier we cannot expect such perfect, instantaneous agreement on group membership. Instead, we ask for an algorithm which satisfies the following properties:

- *(Sa) Agreement on Group Membership*: If two processes are joined to the same group, then they agree on membership of the group.
- *(Sr) Reflexivity*: If a process p is joined to group g, then $p \in members(p)$.
- *(Sm) Monotonicity of Groups*: For any process, the (identifiers of) groups that it joins are strictly increasing in time.
- *(Tj) Bounded Join Delays*: There is a constant J such that if the system is stable at time t, and two processes p and q are connected to each other, then $p \in members(q)$ and $q \in members(p)$ by time $t + JD$.
- *(Td) Bounded Partition Detection Delays*: There is a constant P such that if the system is stable at time t, and two processes p and q are alive but disconnected from each other, then $q \notin members(p)$ and $p \notin members(q)$.
- *(Sl) Stability of Local Views*: If the system is stable at time t, then for any process p, $members(p)$ does not change after $t + \max(J, P)$.
- *(Ti) Bounded Inconsistency Detection Delays*: There is a constant D' such that if process p is joined to group g at time t and $q \in members(p)$, but q doesn't join g by time $t + D'$, then p leaves g by time $t + D'$.

Property Sa is obvious. Sr and Sm are the same as their corresponding properties in Section 3.2. Tj, Td and Sl are the same as the corresponding properties in Section 4.3. Ti says that if p believes it is in some group containing q, but q doesn't join the group for a constant amount of time, then p realizes its error and leaves the group after a constant amount of time.

4.4.1 One Round Protocol

We describe a one round protocol satisfying the above properties. The idea is that when a process detects a membership change, instead of simply adopting a new view by itself, it tells all the other processes about the new view and tries to get them to adopt it too. If the other processes do adopt the new view, then all the processes agree on the group membership, and we do not have the split brains problem. The complication is that different processes may detect different changes to the group membership, and propose different new views. We solve this by having processes timestamp their new view proposals with the time when they observed the membership change. Then when conflicting new views are received, a process adopts the one that's the most up-to-date based on the timestamps.

More precisely, a process p keeps an array of timestamps, with one entry in the array for each process. Every time p sends an "alive" or probe message, it appends a timestamp to it, where the timestamp is obtained by reading p's local clock. Each time p receives an "alive" message with a timestamp from q, it changes the entry for q in its timestamp array to the received timestamp.

If p detects some process failed because p didn't receive an "alive" message on time from its left neighbor q (in the virtual ring), then p suspects q failed and tries to form a new group not including q (note it may not have been q that failed; we'll get to this later). p sends out a "new-group" message with the new view (no q) and p's entire *timestamp array*. If a process p receives a "new-group" message m with view v and timestamp array ta, then it checks ta against its own timestamp array ta'. If all the entries in ta are less than the corresponding entries in ta', then p has more up-to-date information than message m, and so p ignores m. If all entries in ta are more than entries in ta', then p has all out-of-date information compared to m, and so p adopts view v by joining the group for v. If some entries in ta are more than in ta' and some are less, then p finds all the indices where ta is more than ta', and sets the membership on those indices (corresponding to processes) according to v, obtaining a new view v'. Then p proposes a *new* membership with v' as the view, and sends out a "new-group" message with v' and p's timestamp array.

Lastly, when p receives a probe message, it indicates some processes are joining the the current group. Then p performs the above procedure to combine its membership information with that in the probe message, possibly creating a new view including the joining processes.

It is easy to see that the one round protocol satisfies properties Sa, Sr and Sm. We can also see that it satisfies properties Tj, Td, Sl and Ti, because of the following reasons. First, after the system stabilizes, the failure / recovery detectors will continue to trigger new views until the group membership reflects the stabilzied connectivity. Second, the combination of the "alive" messages (for detecting failed processes) and probe messages (for adding recovered processes) will eventually correctly determine the failed and non-failed processes. Lastly, when the failed and non-failed processes are correctly determined, this information will carry the newest timestamp, and since the one round protocol always merges in the newest information, it will eventually propose and adopt correct new views. We don't give a full proof or analysis here; see Appendix B in the paper. However, the time for convergence is slow: $J = \max(\pi + n\delta, \mu) + 5\delta$, $D = \max(\pi + n\delta, \mu) + 5\delta + (n-1)(\pi + (n+1)\delta)$ (note the n^2 term), $D' = 2(\pi + n\delta)$.

4.5 Majority Group Membership Service, Weak Agreement

The one round protocol has some problems. One is the slow stabilization time D, which has an n^2 term. This comes about because of *spurious timeouts*, in which two processes p and q are connected to each other, $q \in members(p)$ but $p \notin members(q)$. As a result, p expects an "alive" message from q, but q doesn't send this message, so p incorrectly tries to form a new group without q, wasting time.

The second problem is that a correct processor can sometimes not detect a quick failure and recovery by another processor. We won't discuss the details of when this happens, but basicaly, if a processor quickly fails and recovers at the same time that a long-failed processor is recovering, then the recovery of the long-failed processor can "mask" the short failure. The reason this is a problem is because a failed process loses its volatile state and sometimes requires a state transfer. So for some applications, correct processes need to know all processes that failed, to bring them up-to-date. The way we solve this problem is to have each process say what the most recent group it joined was. This way, if a process p belonging to group g quickly failed, recovered and joined another group, p's most recent group would just contain itself (a recovering process always joins a group consisting of itself when it recovers), and other processes can detect it failed. Define $pred(p, g)$ be the last group process p joined before joining group g.

In addition to knowing which processes failed, it's useful to know the *order* in which they failed. One way to do this is to guarantee that there is a *linear history* of groups, i.e. a total order on the groups formed in an execution. This requirement is not achievable in an asynchronous environment, but something weaker is. Say that a group is *completed* if all its members eventually join it, and say a group is a *majority group* if it contains a majority of the processes. We require that there is a linear history of *completed majority groups*. If in some application, processes are only allowed to make changes when they belong to a complete, majority group, then in such an application, we know the exact order in which failures and recoveries occur: it is the order they occur in the linear history. Let $Pred(g)$ be the group before g in the linear history (of some execution).

To improve the one round protocol, we want a new protocol with all the properties in Section 4.4, and these additional ones:

- *(Sd) Legitimacy of Processor Deletions*: If process p joins a group g at time t, and $q \notin members(g)$, and q was a member of $pred(p, g)$, there must be a live process r which was not connected to q some time before t.
- *(Sj) Legitimacy of Processor Additions*: If process p joins a group g at time t, and $q \in members(g)$, and q was not a member of $pred(p, g)$, there must be some live process $r \in members(g)$ connected to q some time before t.
- *(Ss) Join Synchronization*: If process p joins group g at time t, then no member of g will be joined to a group $g' < g$ after t.
- *(Sp) Logical Partition Notification*: When process p joins a group g, the protocol should tell p the last group $pred(g, q)$ of all processes $q \in members(g)$.
- *(Sp') Predecessor Notification for Majority Groups*: When a process p joins a majority group g, the protocol should tell p the majority predecessor group to g, $Pred(g)$.

Properties Sd just says there are no spurious deletions of processors. That is, if p and q were connected in $pred(p, g)$, then p can't decide to form and join a new group g for no reason. Only a process r not connected to q can form g. Property Sj is similar. Property Ss is clear. Property Sp and Sp' are to let p determine which processes failed and recovered, and what the linear history of majority groups is, resp.

4.6 Three Round Protocol

The three round protocol we describe combines the protocols of Sections 9 and 10 in the CS paper.

Basically, whereas in the one round protocol, a process that wants to form a new group just tells other processes what group to join (namely, it's own), in the three round protocol, a process tests which other processes are connected whenever it tries to form a new group, and the members of the new group consists of all processes which respond to the test. This way, if two processes are actually connected, neither process will try to form a new group which excludes the other.

In more detail, each process stores what the last group it joined was, and the last majority group that it knows about. It also has an array for storing what the last group joined by any other process is. When a process determines that membership has changed because it failed to receive an "alive" message, it sends a "new-group" message to all the other processes, appended with the new group's ID. However, the message doesn't contain the membership of the proposed new group; that will be determined later. A process that receive a "new-group" message with group ID higher than its current group ID acknowledges the "new-group" message by sending "accept", appended by the last group this process joined, and the last majority group that it knows about. On the other hand, if the process heard about a higher group being formed, it forwards the information about the higher group to the sender of the "new-group" message. A process which sent a "new-group" message waits 2δ time to collect "accept" messages, after which it sets the new members of the group to be those processes that returned "accept". It also updates its array of what groups other processes joined last, and the last majority group that it knows (the maximum majority group that it heard about in the "accept" messages). Then, it sends a "join" message, with the new group's ID, its members (the accepters), the predecessor groups that it knows for other processes, and the latest majority predecessor group. When a process receives a "join" message, it joins the group if it doesn't yet know of a higher group (being) formed.

We won't argue why this protocol satisfies the properties in Section 4.5; however, this is not very difficult. For example, we note why it satisfies property So. The reason is that there can't be two completed majority groups which are unordered with respect to each other, since these groups have a common member, and this members orders the two groups by the order in which it joins them.

This protocol has $J = \max(\pi + (n+3)\delta, \mu) + 9\delta$, $D = \max(\pi + (n+3)\delta, \mu) + 9\delta$, and $D' = 2(\pi + n\delta)$.

5 Moshe: Group Membership for WANs

From the Keidar, Sussman, Marzullo, Dolev paper.

The CS protocols involve sending many messages between arbitrary pairs of processes, and also uses a single time δ as a parameter for fault detection. Therefore, in a Wide Area Network where communication between some pairs of processes, e.g. processes in the same LAN, is much faster and less variable than communication between pairs of processes in different LANs, the CS protocols will constantly detect membership changes and not stabilize. Moshe is a group membership algorithm designed specifically to perform well in WANs. Moshe has four main features:

1. Moshe avoids delivery of *obsolete* views to reduce the overhead of virtual synchrony.
2. Moshe optimized for the common case where the underlying failure detector is relatively consistent, and terminates in a single message delay time. In fact, in simulations over the Internet this common case occurred 99% of the time.
3. Moshe can use different failure detection mechanisms; it is *portable*.
4. Moshe is built using a client-server design to scale well.

We will focus on features 2 and 4.

5.1 Model

Collection of processes communicating across *reliable*, FIFO, point-to-point asynchronous network.

Assume a typical message delay latency of δ, though actual delay is unbounded.

Our goal is to maintain membership of a set C of *client* processes. However, the clients themselves don't maintain the membership, but rely on a set S of *server* processes. More precisely, each client is assigned to one server. The servers run a protocol among themselves to determine a *view*, the set of clients which are mutually connected at some point in time. Then each server informs the clients it is responsible about this view. Views are labeled with labels from a totally ordered set. One way to assign clients to servers is to have each server be responsible for all the clients in its LAN.

We assume an outside *network event notification service*, which tells the servers about changes to (client) connectivity. Specifically, the NE service generates *notification events (NEs)*, with format $NE(joining, leaving)$. Here *joining* and *leaving* are sets of clients which join (recover) or leave (fail) from the network. When the NE service notices a membership change, it is supposed to send (the same) NE message to all servers. However, it may also send (the same)

NE message to only some servers. Each server has a variable called $NSView$ which is its local view of the connected clients. Each time a server receives an NE, it sets its $NSView$ to $NSView \cup NE.joining \backslash NE.leaving$.

The servers run the membership service, and send clients the following messages:

1. $startChange(startChangeNum, suggestedMemb)$: This tells the client the server is engaged in membership change, and the new view is expected to have memberss $suggestedMemb$.
2. $view(id, members, startChangeNums)$: This gives the client a new view, with label id, members $members$.

The $startChangeNum$ fields in the messages have to do with implementing virtual synchrony using the membership service. We will not discuss virtual synchrony, and will ignore the $startChangeNums$.

5.2 Algorithm Overview

Moshe consists of 3 algorithms: a fast algorithm, a blocking-detection algorithm, and a slow algorithm. The program flow is to first run the fast algorithm, see if this blocks using the blocking-detection algorithm, and if so, run the slow algorithm (which can't block). If the maximum message delay is δ, then the fast algorithm runs in time δ, the slow algorithm runs in time 2δ, and so the entire algorithm runs in time 3δ. This assumes perfect behavior from the NE service and stability of the network. We now describe the three algorithms.

5.2.1 Fast Algorithm

Each server waits to hear from the NE service. If it gets an NE message, it updates its $NSView$ variable as described above, and sends a *proposal* message to all the other servers with its new $NSView$. Then it waits to hear a proposal message with the same membership from *every other* server. Whenever it hears a proposal from another server, it puts the proposal in a *props* array. When it hears a proposal from every server, i.e. when *props* is filled, the server sends a *view* message to the client. Then it clears its *props* array, and goes back to listening for NE messages. When it clears the proposal from a process in the *props* array, we say that it *uses* that proposal.

It's easy to see that if when there is a membership change, the NE service informs all the servers of the change, then all the servers will send proposal messages to each other, and the fast algorithm terminates in δ time, if the network is well-behaved.

5.2.2 Blocking Detection Algorithm

The fast algorithm might not terminate if the NE service didn't inform all servers of a membership change. This is because servers which didn't hear about a membership change won't send proposals, and so servers that did hear about the membership change are blocked waiting for proposals from some servers.

Notice that this problem only occurs when some servers get more NE messages than others and does more proposals than some other servers. Thus, the way to detect blocking is to use an array called $usedProps$, where each entry in the array for a server equals the last $propNum$ *used* from that server. When proposals are sent, $usedProp$ is sent with the proposal.

Suppose server p has sent n proposals, and then p gets a proposal $prop$ with an appended $usedProps$ from q in which $usedProps[p] = n$, i.e. q has already used the last proposal which p sent out. Then q must be waiting for a proposal from p, and if p doesn't send a proposal, q is blocked. Thus, the blocking detection algorithm for process p detects blocks by comparing the $usedProps[p]$ of an incoming proposal message against its own $propNum$. If these two numbers are equal, p detects blocking.

5.2.3 Slow Algorithm

Once blocking of the fast algorithm is detected, the slow algorithm starts to run. A server p running the slow algorithm first sets its $propNum$ to be greater than its current $propNum$ and the $propNum$ it heard from any other server. Then it sends a proposal of type SA (slow agreement) to the other servers with this $propNum$. The server might receive some SA proposals itself, in which case it sets its $propNum$ to be equal to the max between its $propNum$ and the $propNum$ in the received proposal, then forwards the proposal with the new $propNum$.

We briefly argue why the slow algorithm terminates. We first note that we only require termination when the network is stable, i.e. when the NE service stops outputting NE events. After this happens, the final NE output by the NE service will be a correct view of the clients' connectivity. It only remains for the servers to agree to this view, by agreeing to each other's (SA) proposals. But this will eventually happen, because the SA proposal with the highest $propNum$ will eventually be adopted by all the servers.

6 Conclusion

We have described the specifications for a group communication service, including safety properties for its membership and multicast algorithm, ordering and reliability constraints on the multicast, and liveness properties. We then described several algorithms for maintaining group membership in a LAN, then an algorithm for maintaining group membership in a WAN. In the next lecture, we will see some algorithms for doing multicast in a GCS, as well as some algorithms built on top of a GCS with membership and multicast services.