A Formal Specification of a Group Communication Service for FLARE (draft) TCD-CS-2001-46

Niels Reijers, Vinny Cahill

17th December 2001

Abstract

In this report we present a formal specification of a group communication service for use in a wireless ad hoc network where there is a high risk of partition. The envisaged GCS makes use of a partition anticipator and allows separate partitions to make progress when the partition was predicted by this partition anticipator. In case of unanticipated partitions or failures, only the majority partition is allowed to continue, while members of minority partitions are dropped from the group.
A partitioned state indicator is provided to inform the application whether the group is in an unpartitioned state or whether there may be members in another partition. A timeout is used to ensure that the group will eventually return to an unpartitioned state.
Predicate logic is used to give a detailed formal specification of the GCS and its behaviour in the presence of partitions.
This document represents an initial draft version of the specification and is expected to contain inaccuracies. It is provided only to solicit feedback from interested readers.

Contents

1 Introduction

We will use the specification method used in [Vit01] to specify the semantics of our group communication service. The correct behaviour of the group communication service will be specified using 'properties' that limit the set of legal sequences of actions, called 'traces'. Various definitions will be made to make things more readable.
We have copied many definitions and properties from [Vit01] that apply to our service, some have been modified for our specific needs. These are all marked with a *.

2 The Model

We will use the following types, partially from [Vit01]:

Types

$\mathcal{P}$ The set of process identifiers.

$\mathcal{M}$ The set of messages sent by the application.

$\mathcal{VID}$ The set view identifiers, partially ordered by the $<$ operator.

In addition we will also use these types, functions and constants:

$\mathcal{PID}$ The set partition identifiers, partially ordered by the $<$ operator.

$timeout \in \mathbb{R}$ The group's timeout value.

$time\_of(t_{\mathbb{N}}) \rightarrow \mathbb{R}$ A surjective function returning the time at which an action occurred.

$pid(t_{\mathbb{N}}) \rightarrow \mathcal{P}$ A surjective function returning the process at which an action occured.

Actions Figure 2 illustrates the actions of the group communication, and whether they are input or output actions. The signature for each action is listed below. Each action has at least one parameter, indicating the process where the action took place.

- input $send(p, m), p \in \mathcal{P}, m \in \mathcal{M}$
- output $recv(p, m), p \in \mathcal{P}, m \in \mathcal{M}$
- output $safe\_prefix(p, m), p \in \mathcal{P}, m \in \mathcal{M}$
- input $create(p), p \in \mathcal{P}$
- input $join(p, q), p \in \mathcal{P}, q \in \mathcal{P}$

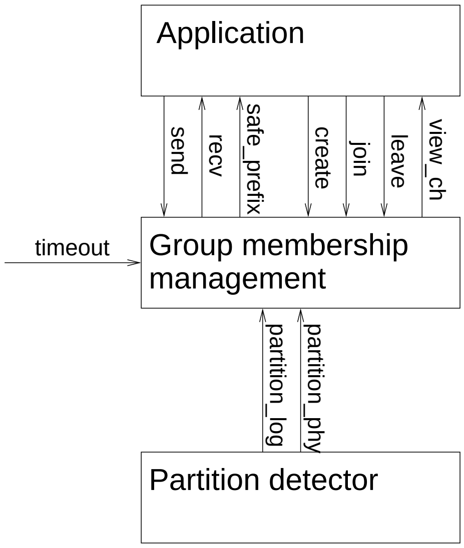

Figure 1: The external actions of the group communication service

- input $leave(p), p \in \mathcal{P}$
- output $view\_ch(p, \langle id, members, part\rangle, T), p \in \mathcal{P}, id \in \mathcal{VID}, members \in 2^{\mathcal{P}}, part \in \{true, false\}, T \in 2^{\mathcal{P}}$
- input $timeout(p, q), p \in \mathcal{P}, q \in \mathcal{P}$
- input $partition\_log(p, \langle id, processes, main\rangle), p \in \mathcal{P}, processes \in 2^{\mathcal{P}}, main \in \{true, false\}$
- input $partition\_phy(p, \langle id, processes, main\rangle), p \in \mathcal{P}, processes \in 2^{\mathcal{P}}, main \in \{true, false\}$

A node p needs to be in contact with at least 1 member, q, to join.
The group view consists of a view identifier, id, a list of *members* in the current view, $members$, the partitioned indicator, and the transitional set, T, containing the subset of members in the new view that had the same previous view as p. When the partitioned indicator is false, all members are in the current view.
A partition action indicates that from that point on, communication is only possible with the *processes* (not limited to group members) in *processes*.
The timeout action indicates that process p is notified of the fact that process q has timed out.
The $partition\_log$ action announces an anticipated partition, and the $partition\_phy$ action that physical communication with nodes that aren't in the *processes* list has become impossible. Whether this is unanticipated or not depends on whether there was a preceding $partition\_log$ event that annouced the disconnection.

For both partition actions, the *main* boolean indicates which partition is the main partition.

3 Basic definitions and properties

We start with some straight forward definitions and properties. The definition $leave\_at(p,i)$ will later be replaced by a more complex one. A process joins the group either by a join action, or by creating it (section 4).

Definition 1 (Join at, leave at, partition at, and any action)

$$
\begin{array}{ll}
join\_at(p,i) & \stackrel{\text{def}}{=} \exists q(t_i = join(p,q)) \\
 & \vee t_i = create(p) \\
join\_at(p,q,i) & \stackrel{\text{def}}{=} t_i = join(p,q) \\
leave\_at(p,i) & \stackrel{\text{def}}{=} t_i = leave(p) \\
partition\_at(p,P,i) & \stackrel{\text{def}}{=} t_i = partition\_log(p,P) \\
 & \vee t_i = partition\_phy(p,P) \\
any\_action(p,i) & \stackrel{\text{def}}{=} \exists m(t_i = send(p,m)) \\
 & \vee \exists m(t_i = recv(p,m)) \\
 & \vee \exists q(t_i = join(p,q)) \\
 & \vee t_i = leave(p)) \\
 & \vee t_i = create(p)) \\
 & \vee \exists V,T(t_i = view\_ch(p,V,T)) \\
 & \vee \exists P(t_i = partition\_phy(p,P)) \\
 & \vee \exists P(t_i = partition\_log(p,P)) \\
 & \vee \exists q(t_i = timeout(p,q))
\end{array}
$$

Further, we copy some basic definitions from [Vit01], and define some equivalent shorthands for partitions. A formal definition of $view\_of$ and $partition\_of$ will be given in section 5.1. For now it suffices to say that they return the last delivered view/partition at the node where t_i occurs.

Definition 2 (Basic definitions*)

$$
\begin{array}{lll}
receives(p,m) & \stackrel{\text{def}}{=} & \exists i(t_i = recv(p,m)) \\
receives\_in(p,m,V) & \stackrel{\text{def}}{=} & \exists i(t_i = recv(p,m) \wedge view\_of(t_i) = V) \\
recv\_before(p,m,m') & \stackrel{\text{def}}{=} & \exists i,j(t_i = recv(p,m) \wedge t_j = recv(p,m') \wedge i < j) \\
recv\_before\_in(p,m,m',V) & \stackrel{\text{def}}{=} & \exists i,j(t_i = recv(p,m) \wedge t_j = recv(p,m') \\
 & & \wedge view\_of(t_i) = view\_of(t_j) = V \wedge i < j) \\
sends(p,m) & \stackrel{\text{def}}{=} & \exists i(t_i = send(p,m)) \\
sends\_in(p,m,V) & \stackrel{\text{def}}{=} & \exists i(t_i = send(p,m) \wedge view\_of(t_i) = V) \\
installs(p,V) & \stackrel{\text{def}}{=} & \exists i(t_i = view\_ch(p,V,T)) \\
installs\_in(p,V,V') & \stackrel{\text{def}}{=} & \exists i(t_i = view\_ch(p,V,T) \wedge view\_of(t_i) = V') \\
partitions(p,P) & \stackrel{\text{def}}{=} & \exists i(partition\_at(p,P,i)) \\
partitions\_in(p,P,P') & \stackrel{\text{def}}{=} & \exists i(partition\_at(p,P,i) \wedge partition\_of(t_i) = P')
\end{array}
$$

The following assumption will simplify the presentation of the other properties. It doesn't limit the usefulness of the group communication service, because it is easily satisfied by appending a member id and sequence number to the messages.

Assumption 1 (Message uniqueness*) *There are no two different send events with the same content.*
Formally:

$$t_i = send(p, m) \wedge t_j = send(q, m) \Rightarrow i = j$$

Finally, we assume that the $time\_of(t_i)$ function returns increasing time values.

Assumption 2 (Increasing time) *We require that when an event occurs after another, it has a higher time_ of value.*
Formally:

$$i < j \Rightarrow time\_of(t_i) < time\_of(t_j)$$

Property 1 (tmp) *This is just here to keep the numbering consistent with previous versions.*

3.1 Membership

We will define group membership in terms of the $join\_at$ and $leave\_at$ shorthands. The latter will be redefined in section 10.

Definition 3 (Membership) *A node is a member in the time between after its join and up to and including its leave event.*
Formally:

$$\begin{aligned} member\_at(p,i) \quad & \stackrel{\text{def}}{=} \exists j(j < i \wedge join\_at(p,j) \\ & \wedge \neg\exists k(j < k < i \wedge leave\_at(p,k))) \end{aligned}$$

Definition 4 (Membership after)

$$member\_after(p,i) \stackrel{\text{def}}{=} \forall j(j \geq i \Rightarrow member\_at(p,j))$$

We will require that a process will only become a member once. This may seem like a limitation, but in fact it just means that when a process leaves the group, and wants to join again, it will have another identifier in the group. So p and q, both in $\mathcal{P}$, may represent the same process in reality.
Therefore, the following property doesn't limit the use of the group communication service, but it will make the specification much more simple.

Property 2 (Single membership) *A process is only a member* once.
Formally:

$$\begin{aligned} \forall i,j(\quad & member\_at(p,i) \wedge member\_at(p,j) \wedge i < j \\ & \Rightarrow \forall k(i < k < j \Rightarrow member\_at(p,k))) \end{aligned}$$

We further require a process to be a member of the group in order to install a group view or to receive messages.

Property 3 (Actions requiring membership) *Installing a new view and receiving messages and receiving timeouts require membership.*
Formally:

$$t_i = view\_ch(p,V) \vee t_i = recv(p,m) \vee t_i = timeout(p,q) \Rightarrow member\_at(p,i)$$

To join the group, a process must be in contact with at least one process who is either a member or a *designated join point* of the group. The designated join point will be defined in section 11.

Property 4 (Join requirement)

$$\begin{aligned} join\_at(p,q,i) \Rightarrow \quad & (member\_at(q,i) \vee djp\_at(q,i)) \\ \wedge \quad & p \in partition\_of(t_i,q) \\ \wedge \quad & q \in partition\_of(t_i,p) \end{aligned}$$

4 Group creation

Group creation is quite straightforward. The group is created using the $create(p)$ action. When a process creates the group, it immediately becomes the first member of the group, as specified in Definition 1 ($join\_at(p,i)$).
Because we only concern ourselves with one group in this document, we require that the group be created only once.

Property 5 (A group is created only once)

$$t_i = create(p) \Rightarrow \neg\exists j, q(t_j = create(q) \wedge j \neq i)$$

We also need to make sure that at first the main partition is the one containing the process that created the group. Before the group was created there can be no main partition.

Property 6 (No main partition before create action)

$$partition\_of(t_i).main = true \Rightarrow \exists j, p(j < i \wedge t_j = create(p)$$

When the group is created, the partition becomes the main partition:

Property 7 (Main partition creation) *After creating the group, the next action is installing the main partition.*
Formally:

$$\begin{array}{rl} t_i = create(p) \Rightarrow & \exists j, P(next\_action(i,j) \\ \wedge & t_j = partition\_phy(p,P) \\ \wedge & P.main = true) \end{array}$$

with

Definition 5 (Next action)

$$next\_action(i,j) \stackrel{\text{def}}{=} \neg\exists k, q(i < k < j \wedge any\_action(q,k))$$

5 Group view and partitions

In this section we will describe how the group view and partitions relate, and what the consequences of partitions are.

5.1 General properties

First some general properties, most of which hold for both partitions and group views.

Property 8 (Self inclusion*) *When a group view or partition is installed, this will always contain the node that it is installed on.*
Formally:

$$installs(p, V) \Rightarrow p \in V.members$$

and

$$partitions(p, P) \Rightarrow p \in P.members$$

Property 9 (Local monotonicity*) *When a node installs two group views or partitions, the one that is installed last has a higher id than the first.*
Formally:

$$t_i = view\_ch(p, V, T) \wedge t_j = view\_ch(p, V', T') \wedge i < j \Rightarrow V.id < V'.id$$

and

$$partition\_at(p, P, i) \wedge partition\_at(p, P', j) \wedge i < j \Rightarrow P.id < P'.id$$

Property 10 (Untitled) *When a node installs a group view or partition, all members in that group view or partition will install the same.*
Formally:

$$installs(p, V) \Rightarrow \forall q(q \in V.members \Rightarrow installs(q, V))$$

and

$$partitions(p, P) \Rightarrow \forall q(q \in P.members \Rightarrow partitions(q, P))$$

The following property only holds for the group view. A partition contains all *processes* with which communication is possible, whether they are a member or not.

Property 11 (Group views contain only members) *Formally:*

$$t_i = view\_ch(p, V) \Rightarrow \forall q(q \in V.members \Rightarrow member\_at(q, i))$$

Property 12 (Initial view event*) *Every recv event occurs within some view.*
Formally:

$$t_i = recv(p, m) \vee t_i = send(p, m) \Rightarrow view\_of(t_i) \neq \perp$$

5.2 The Transitional Set

Property 13 (Transitional Set*) *The transitional set is a subset of the intersection between the member sets of the current and previous view.*
Formally:

$$t_i = view\_ch(p, V, T) \wedge view\_of(t_i) = V' \Rightarrow T \subseteq V.members \cap V'.members$$

If p and q install the same view, q is included in p's transitional set if and only if p's previous view was also identical to q's previous view.
Formally:

$$\begin{array}{l} t_i = view\_ch(p, V, T) \wedge view\_of(t_i) = V' \wedge installs\_in(q, V, V'') \\ \Rightarrow (q \in T \Leftrightarrow V' = V'') \end{array}$$

5.3 View of/Partition of

Since we will be using partitions and group views, we introduce two definitions that give us the current partition or group view of an event.

Definition 6 (partition of) *The partition of an action t_i for process p is the partition delivered with the last partition event before t_i at p.*
Formally:

$$partition\_of(t_i, p) \stackrel{\text{def}}{=} \begin{cases} P & if & \exists j, P(j < i \wedge partition\_at(p, P, i) \\ & \wedge & \neg\exists k, P'(j < k < i \wedge partition\_at(p, P, j))) \\ \perp & otherwise & \end{cases}$$

The partition of an action t_i is the partition of t_i at the node where the action took place:

$$partition\_of(t_i) \stackrel{\text{def}}{=} partition\_of(t_i, pid(t_i))$$

Definition 7 (view of*) *The view of an action t_i for process p is the view installed in the last view_ch event before t_i at p.*
Formally:

$$view\_of(t_i, p) \stackrel{\text{def}}{=} \begin{cases} P & if & \exists j, P, T(j < i \wedge t_j = view\_ch(p, P, T) \\ & \wedge & \neg\exists k, P', T'(j < k < i \wedge t_k = view\_ch(p, P', T'))) \\ \perp & otherwise & \end{cases}$$

The partition of an action t_i is the partition of t_i at the node where the action took place:

$$view\_of(t_i) \stackrel{\text{def}}{=} view\_of(t_i, pid(t_i))$$

5.4 Partitions

A partition event indicates that the subset of processes with which communication is possible has changed. This means that if a new group view is installed, we should not include members that are not in the latest partition. To avoid trivial implementations where each node installs a group view containing just itself, we also require that a group view contains *all* the processes in the lastest partition that are members.

Property 14 (Group view membership)
Formally:

$$\begin{aligned} t_i = view\_ch(p,V,T) \wedge \quad & partition\_of(t_i) = P \\ \Rightarrow \quad & \forall q(member\_at(q,i) \\ \wedge \quad & q \in P.members \\ \Leftrightarrow \quad & q \in V.members) \end{aligned}$$

This property implies that two nodes in different partitions can never install a group view which includes them both.
Finally, we require that the first action at any process is a partition_phy so it has information on the network status.

Property 15 (First action) *The first action any node takes is a partition_phy action.*

$$\begin{aligned} & any\_action(p,i) \wedge \neg\exists P(t_i = partition\_phy(p,P)) \\ \Rightarrow \quad & \exists j,q,P(j < i \wedge t_j = partition\_phy(p,P)) \end{aligned}$$

5.5 Partitioned indicator

Remember the signature of the view_ch action:

$$view\_ch(p, \langle id, members, part \rangle, T)$$

with

$$part \in \{true, false\}$$

The partition boolean indicates whether there are members that are not included in the current group view. We require that if this boolean is false we know that there are no members outside the view. The reverse cannot be guaranteed, since it is impossible to implement.

Property 16 (Unpartitioned state) *When the partition indicator is false, all members are in the current view.*
Formally:

$$\begin{aligned} & t_i = view\_ch(p,V,T) \\ \wedge \quad & V.part = false \\ \Rightarrow \quad & \forall q(is\_member\_at(q,i) \Rightarrow q \in V.members) \end{aligned}$$

The timeout option of our policy guarantees that if no more partitions occur, we can eventually draw the conclusion that there are no more members outside the group view.

Property 17 (Partition recovery A) *When no anticipated partition occurs for timeout time, the group is in the unpartitioned state.*

$$\begin{array}{ll} & any\_action(p,i) \\ \wedge & \neg\exists j,P(j<i \\ \wedge & anticipated_partition(p,P,j) \\ \wedge & time\_of(t_i) - time\_of(t_j) < timeout) \\ \Rightarrow & \exists V,T((view\_of(t_i)=V \;\vee\; t_i = view\_ch(p,V,T)) \\ \wedge & V.part = false) \end{array}$$

We will also require that if a node could have gathered enough information to conclude the network is unpartitioned, he will do so. Members in a group view will see other members join and leave. They can also communicate this information to other members. We define two shorthands to determine if a node has seen a member leave or join, or has heard about these events from other members.
The first determines if a member p, has at time t_i knowledge about the fact that member q left at time t_j. It can know this either by being in the view when q left, or because it heard about this from another member.
We include three other ways of leaving the group, through timeouts and minority partition, and through minority partition recovery, so that we won't have to redefine *seen_leave_at* when we introduce these concepts in sections 7 and 9.

Definition 8 (Seen leave at) *At time t_i, process p is aware of the fact that process q has left the group.*
Formally:

$$\begin{array}{rl} seen\_leave(p,q,i) \stackrel{\text{def}}{=} & \\ (member\_at(p,i) \vee djp\_at(p,i)) \wedge (& \\ & \exists j(j<i \\ & \wedge(t_j = leave(q) \\ & \vee t_j = timeout(q,q) \\ & \vee dropped\_maj(q,j)) \\ & \wedge\exists P(partition\_of(t_j,p) = P \wedge q \in P.members \\ & \wedge(member\_at(p,j) \vee djp\_at(p,j)))) \\ \vee & \\ & know\_dropped\_minority(p,q,i) \\ \vee & \\ & \exists k,P',p'(k<i \\ & \wedge partition\_of(t_k,p) = P' \\ & \wedge p' \in P'.processes \\ & \wedge seen\_leave(p',q,k) \\ & \wedge(member\_at(p,k) \vee djp\_at(p,k)))) \\ ) & \end{array}$$

And we give a similar definition for joining the group.

Definition 9 (Seen join at) *At time t_i, process p is aware of the fact that process q has joined the group.*
Formally:

$$
\begin{array}{rl}
seen\_join(p,q,i) \stackrel{\text{def}}{=} & \\
(member\_at(p,i) \vee djp\_at(p,i)) \wedge (& \\
& \exists q', j, P(j < i \\
& \wedge join\_at(q,q',j) \\
& \wedge partition\_of(t_j, q') = P \\
& \wedge p \in P.members \\
& \wedge (member\_at(p,j) \vee djp\_at(p,j))) \\
\vee & \\
& \exists k, P', p'(k < i \\
& \wedge partition\_of(t_k) = P' \\
& \wedge p \in P'.members \\
& \wedge p' \in P'.members \\
& \wedge seen\_join(p',q,k) \\
& \wedge (member\_at(p,k) \vee djp\_at(p,k))) \\
) &
\end{array}
$$

If the members in a partition know that each process they know to have joined, is present or has left, we require the partition indicator to be false.

Property 18 (Partition recovery B)

$$
\begin{array}{rl}
any\_action(p,i) \wedge & (V = view\_of(t_i) \vee t_i = view\_ch(p,V)) \\
\Rightarrow (& \\
\forall q(& \\
& q \in V.members \\
\vee & \\
& \neg \exists p'(p' \in V.members \\
& \wedge seen\_join(p',q,i)) \\
\vee & \\
& \exists p'(p' \in V.members \\
& \wedge seen\_leave(p',q,i) \\
) & \Rightarrow V.part = false \\
) &
\end{array}
$$

The reason this is implementable is that if a proces joins, some member is informed about this. This means if the group enters the partitioned state, some members are missing in the partitions. These members will remain missing until they merge again. If in the mean time they let new processes join, they will inform the other partition of this if all the original members merge again. This way, when there is a partition, there will always be some members missing, and from that it follows that if there are no members missing, there is no partition. When there are members missing, this does *not* imply that there is a partition, all the missing members may have left while they were partitioned. Therefore we cannot require the partition indicator to always be false when there is no partition, since this cannot be implemented.

6 Message delivery

This section will define our message delivery guarantees. We start with a few trivial ones.

6.1 Basic properties

Property 19 (No duplication*) *A message is only delivered once.*
Formally:

$$t_i = recv(p, m) \wedge t_j = recv(p, m) \Rightarrow i = j$$

Property 20 (Delivery integrity*) *For each receive action there is a preceding send action, and the sender was a member at the time of sending.*
Formally:

$$t_i = recv(p, m) \Rightarrow \exists j, q(j < i \wedge t_j = send(q, m) \wedge member\_at(q, j))$$

We require that a message is delivered in one view only.

Property 21 (Same view delivery*) *If p and q both receive m, they receive m in the same view.*
Formally:

$$receives\_in(p, m, V) \wedge receives\_in(q, m, V') \Rightarrow V = V'$$

Property 22 (Own messages) *A node that sends a message and stays a member, eventually receives its own message.*
Formally:

$$t_i = send(p, m) \wedge member\_after(p, i) \Rightarrow receives(p, m)$$

Finally, we need a property that actually disallows communication between partitions. Right now it would be legal for a message sent in one partition to be delivered in another partition.
When the sender remains a member Properties 21 and 22 guarantee that it is only delivered in the partition the sender is in, but when the sender leaves the group, we don't want the message to be delivered in a different partition.

Property 23 (Delivery in one view only) *A message is delivered only in one view.*
Formally:

$$\begin{aligned} sends(p, m) \wedge t_i = recv(q, m) \Rightarrow \quad & p \in view\_of(t_i, q).processes \\ \wedge \quad & q \in view\_of(t_i, p).processes \end{aligned}$$

This property is actually a bit strickter than just preventing communication between partitions. A node could split from the sender's partition and join another partition to deliver the message there. This can happen without communication between the partitions, but is excluded with this property. However the restriction also follows from the same view delivery property.

6.2 Delivery guarantees

Currently, we have two options for the message delivery guarantee. The first is a weak atomic multicast, the second is virtual synchrony. Together with same view delivery, atomicity guarantees virtual synchrony.

6.2.1 Option A for delivery guarantee

Property 24 (Weak atomicity) *We require that if a message is received by one member, it will be received by all members in that member's current view that will install a new view.*
Formally:

$$\begin{aligned} receives\_in(p,m,V) \Rightarrow\ & \forall q(q \in V.members \\ \wedge\ & \exists V'(installs\_in(p,V',V)) \\ \Rightarrow\ & receives\_in(q,m,V)) \end{aligned}$$

6.2.2 Option B for delivery guarantee

This property follows from the Atomicity and same view delivery properties. If we go with those, we won't need Virtual Synchrony.

Property 25 (Virtual Synchrony*) *If processes p and q install the same new view V in the same previous view V', then any message received by p in V' is also received by q in V'.*
Formally:

$$\begin{aligned} & installs\_in(p,V,V') \wedge installs\_in(q,V,V') \wedge receives\_in(p,m,V') \\ & \Rightarrow receives\_in(q,m,V') \end{aligned}$$

We will also want the safe message indication if we choose option C:

Definition 10 (Safe indication and stable messages*) *A message m received in a view V is indicated as safe at process p:*

$$\begin{aligned} indicated\_safe(p,m,V) \stackrel{\text{def}}{=}\ & receives\_in(p,m,V) \\ \wedge\ & \exists i(t_i = safe\_prefix(p,m) \\ & \vee \exists m'(t_i = safe\_prefix(p,m') \\ & \wedge recv\_before\_in(p,m,m',V))) \end{aligned}$$

Message m is stable in view V:

$$stable(m,V) \stackrel{\text{def}}{=} \forall p(p \in V.members \Rightarrow receives(p,m))$$

Property 26 (Safe indication prefix*) *If a message is indicated as safe, then it is stable in the view in which it was received.*
Formally:

$$indicated\_safe(p,m,V) \Rightarrow stable(m,V)$$

Property 27 (Safe indication reliable prefix*) *If message m is indicated as safe at some process p and m is also delivered by process q in view V, then every message delivered at q before m in V is also stable in V.*
Formally:

$$indicated\_safe(p, m, V) \wedge recv\_before\_in(q, m', m, V) \Rightarrow stable(m', V)$$

6.3 Ordering guarantees

We make 3 ordering guarantees available: FIFO, causal and total ordering. For now, there is no guarantee on ordering of messages sent using different guarantees.
The 3 different types are presented here are properties. Of course not all of these properties would have to hold for all messages. For messages sent with FIFO ordering 28 and 29 would have to hold, for causal 30 and 31 and for total ordering 32 and 33.

Property 28 (FIFO*) *If a process p sends two messages, then these messages are received in the order in which they were sent at every process that receives them both.*
Formally:

$$\begin{array}{ll} & t_i = send(p, m) \\ \wedge & t_j = send(p, m') \\ \wedge & i < j \\ \wedge & t_k = recv(q, m) \\ \wedge & t_l = recv(q, m') \\ \Rightarrow & k < l \end{array}$$

Property 29 (Reliable FIFO*) *If p sends m before m' in the same view, then any process that receives m' receives m before m'.*
Formally:

$$\begin{array}{ll} & t_i = send(p, m) \\ \wedge & t_j = send(p, m') \\ \wedge & i < j \\ \wedge & view\_of(t_i) = view\_of(t_j) \\ \wedge & t_k = recv(q, m') \\ \Rightarrow & \exists l (l < k \wedge t_l = recv(q, m)) \end{array}$$

Definition 11 (Causal ordering*) *Causal ordering as defined by Lamport [Lam78]:*

$$\begin{array}{rcl} t_i \to t_j & \stackrel{\text{def}}{=} & (pid(t_i) = pid(t_j) \wedge j \geq i) \\ & \vee & (t_i = send(p, m) \wedge t_j = recv(q, m)) \\ & \vee & \exists k (t_i \to t_k \wedge t_k \to t_j) \end{array}$$

Property 30 (Causal Delivery*) *If two messages m and m' are sent so that m causally precedes m', then every process that receives both these messages,*

receives m before m'.
Formally:

$$\begin{array}{ll} & t_i = send(p,m) \\ \wedge & t_j = send(p',m') \\ \wedge & t_i \rightarrow t_j \\ \wedge & t_k = recv(q,m) \\ \wedge & t_l = recv(q,m') \\ \Rightarrow & k < l \end{array}$$

Property 31 (Reliable Causal*) *If message m causally precedes a message m' and both are sent in the same view, then every process that receives both these messages, receives m before m'. Formally:*

$$\begin{array}{ll} & t_i = send(p,m) \\ \wedge & t_j = send(p',m') \\ \wedge & t_i \rightarrow t_j \\ \wedge & view\_of(t_i) = view\_of(t_j) \\ \wedge & receives(q,m') \\ \Rightarrow & \exists l(l < k \wedge t_l = recv(q,m)) \end{array}$$

Definition 12 (Timestamp function*) *A timestamp (TS) function is a one-to-one function from $\mathcal{M}$ to the set of natural numbers:*

$$TS\_function \stackrel{\text{def}}{=} f : \mathcal{M} \rightarrow \mathbb{N} \wedge f(m) = f(m') \Rightarrow m = m'$$

Definition 13 (Last view*) *V is the last view installed at process p if p installs V and does not install any views after V.*
Formally:

$$\begin{array}{lll} last\_view(p,V) & \stackrel{\text{def}}{=} & \exists i, T(t_i = view\_ch(p,V,T) \\ & \wedge & \neg\exists j, T', V'(j > i \\ & \wedge & t_j = view\_ch(p,V',T'))) \end{array}$$

Property 32 (Weak Total Order*)
1: For every pair of views V and V' there is a TS function f so that every process that installs V in V' receives messages in V' in an order consistent with f.
Formally:

$$\begin{array}{l} \forall V, V' \exists f(TS\_function(f) \\ \wedge \forall p, m, m'((installs\_in(p,V,V') \wedge recv\_before\_in(p,m,m',V') \\ \quad \Rightarrow f(m) < f(m')))) \end{array}$$

2: For every view V there is a TS function f so that every process that has V as its last view receives messages in V in an order consistent with f.
Formally:

$$\begin{array}{l} \forall V, V' \exists f(TS\_function(f) \\ \wedge \forall p, m, m'((last\_view(p,V) \wedge recv\_before\_in(p,m,m',V') \\ \quad \Rightarrow f(m) < f(m')))) \end{array}$$

Property 33 (Reliable Total Order*) *There is a timestamp function f such that if q receives m', and messages m and m' were sent in the same view with f(m)<f(m'), then q receives m before m'.*
Formally:

$$\begin{array}{rl}
\multicolumn{2}{l}{\exists f(TS_function(f) \wedge \forall V, m, m', p, p', q(} \\
& sends_in(p, m, V) \\
\wedge & sends_in(p', m', V) \\
\wedge & receives(q, m') \\
\wedge & f(m) < f(m') \\
\Rightarrow & recv_before(q, m, m')))
\end{array}$$

7 Anticipated/unanticipated and minority/majority partitions

In the following sections we will include failures in our model, and refine our notion of membership. In the previous sections it was simply assumed that there was a leave primitive which was the only way of leaving the group. In these sections we will change the definition of leaving the group to include timeouts and unanticipated partitions.
We start by defining what an anticipated and an unanticipated partition is in terms of *partition_phy* and *partition_log* actions.

Definition 14 (anticipated partitions) *Any partition_log action is an anticipated partition. Formally:*

$$anticipated\_partition(p, P, i) \stackrel{\text{def}}{=} t_i = partition\_log(p, P)$$

Definition 15 (anticipated partition of) *This works just like partition_of:*

$$anticipated\_partition\_of(t_i, p) \stackrel{\text{def}}{=}$$
$$\begin{cases} P & if \quad \exists j, P(j < i \wedge anticipated\_partition(p, P, i) \\ & \wedge \quad \neg\exists k, P'(j < k < i \wedge anticipated\_partition(p, P, j))) \\ \perp & otherwise \end{cases}$$

and

$$anticipated\_partition\_of(t_i) \stackrel{\text{def}}{=} anticipated\_partition\_of(t_i, pid(t_i))$$

Since the *partition_log* serves to announce upcoming partitions, not all *partition_phy* actions are unanticipated partitions. The only partitions that are unanticipated are the physical partitions that happen within $safe$ time of the last anticipated partition, or that exclude members that were in the last anticipated partition.
Since partitions contain all processes, and not just members, we need to determine which processes were members in the last anticipated partition. We want to determine this as late as possible, and choose the time of the first installation of the new partition.
At this point we should realise that different processes in the new partition may have different previous partitions, so taking the first installation of the new partition may be inaccurate. Instead we take the first installation of the new partition in the previous one.
We define a shorthand that gives the time when the first new partition was installed in a partition:

Definition 16 (End partition) *This shorthand give the first i such that t_i is the installation of a new partition in P.*

Formally:

$$end(P) \stackrel{\text{def}}{=} \begin{cases} i & if \quad \exists i, P', p'(partition(p', P', i) \wedge partition\_of(t_i) = P \\ & \quad \wedge \neg\exists j, P'', p''(partition(p'', P'', j) \wedge partition\_of(t_j) = P \wedge j < i)) \\ \perp & otherwise \end{cases}$$

So for an action $t_i = partition\_phy(p, P)$, the time to determine membership is $end(partition\_of(t_i))$.
We define a function to select the members of a partition at a certain time:

Definition 17 (Members in)

$$members\_in(P, i) \stackrel{\text{def}}{=} \{p, p \in P.processes \wedge member\_at(p, i)\}$$

Using these definitions we can determine whether a partition is unanticipated:

Definition 18 (unanticipated partitions) *A partition_phy is only an unanticipated partition when the last partition_log was less than safe time ago, or included nodes that are now unreachable. Formally:*

$$\begin{array}{rl} \multicolumn{2}{l}{unanticipated\_partition(p, P, i) \stackrel{\text{def}}{=}} \\ t_i = & partition\_phy(p, P) \\ \wedge & \neg\exists j, P'(j < i \\ \wedge & t_j = partition\_log(p, P') \\ \wedge & time\_of(t_i) - time\_of(t_j) > safe \\ \wedge & P' = anticipated\_partition\_of(t_i) \\ \wedge & \forall q(q \in members\_in(P', end(P')) \rightarrow q \in P)) \end{array}$$

Knowing which partitions are anticipated and which aren't, the definitions appear straightforward, but again we need to be careful about determining the members in a partition. We count the number of members in the current partition at the time the new partition is first installed at any member (the time defined by $end(P)$). In combination with the new definition of *leave_at* that will be introduced in section 10, some members may already have been dropped from the group when the members are counted independently by some process. However, this can only happen in a partition that was a minority partition to begin with. So although the count may not be accurate, the result will be the same.
For the members in the original partition, we count the same members as we did in the definition of *unanticipated_partition*.

Definition 19 (Minority partition) *An unanticipated partition is a minority partition if less than or equal to half of the members in the previous partition* *are present. Formally:*

$$
\begin{array}{rcc}
minority\_partition(p, P, i) & \stackrel{\text{def}}{=} & unanticipated\_partition(p, P, i) \\
& \wedge & \exists P'(partition\_of(t_i) = P' \\
& \wedge & 2\,|members\_in(P, i) \cap members\_in(P', end(P')| \\
& \leq & |members\_in(P', end(P'))|
\end{array}
$$

Definition 20 (Majority partition) *An unanticipated partition is a majority partition if it is not a minority partition. Formally:*

$$
\begin{array}{rcl}
majority\_partition(p, P, i) & \stackrel{\text{def}}{=} & unanticipated\_partition(p, P, i) \\
& \wedge & \neg minority\_partition(p, P, i)
\end{array}
$$

8 The main partition

Some formal definitions and properties for our concept of a main partition.

Property 34 (At most one main partition) *When a partition splits into more than one different partitions, only one can be the main partition.*

$$\begin{array}{cc} & partitions\_in(p, P, P'') \\ \wedge & partitions\_in(q, P', P'') \\ \wedge & P.main = P'.main = true \\ \Rightarrow & P = P' \end{array}$$

Property 35 (At least 1 main partition) *When the main partition splits, there will be a new main partition.*

$$\begin{array}{cc} & partitions\_in(p, P, P') \\ \wedge & P'.main = true \\ \Rightarrow & (\exists q, P''(partitions\_in(q, P'', P') \\ \wedge & P''.main = true) \end{array}$$

Property 36 (A main partition can only be formed from the main partition) *When a partition is installed that is the main partition, at least one of its members must have come from the old main partition.*

$$\begin{array}{cc} & partitions(p, P) \\ \wedge & P.main = true \\ \Rightarrow & \exists q, j, P'(q \in P.processes \\ \wedge & partitions\_in(q, P, P') \\ \wedge & P'.main = true) \end{array}$$

Property 37 (Main partition after an unanticipated partition goes to the majority) *If an unanticipated partition occurs in the main partition, the new main partition should be the majority, if it exists.*

$$\begin{array}{cl} & unanticipated\_partition(p, P, i) \\ & partitions\_in(p, P, P'') \\ \wedge & P.main = true \\ \Rightarrow & majority\_partition(p, P, i) \\ \vee & (\neg\exists q, P', j(partitions\_in(p, P', P'') \\ & \wedge majority\_partition(q, P', j)) \end{array}$$

If possible, the main partition must be chosen in such a way that there is a member in it.

Property 38 (Members in main partition) *If a main partition that has members splits, the result must also have members. The meaning of djp_at will be explained in section11.*
Formally:

$$
\begin{array}{ll}
 & partition\_at(p, P, i) \\
\wedge & partitions\_in(p, P, P'') \\
\wedge & P''.main = true \\
\wedge & \neg\exists q(q \in P''.processes \\
 & \wedge(member\_at(q, i) \vee djp\_at(q, i))) \\
\Rightarrow & \neg\exists p', P', j(partition\_at(p', P', j) \\
\wedge & partitions\_in(p', P', P'') \\
\wedge & (member\_at(p', i) \vee djp\_at(p', i)))
\end{array}
$$

Definition 21 (in_main_partition_at) *A member is in the main partition if its last partition event indicated this. Formally:*

$$in\_main\_partition\_at(p, i) \stackrel{\text{def}}{=} partition\_of(t_i, p).main = true$$

9 Timeouts

When a partition is disconnected from the main partition for a certain amount of time, a timeout occurs. This timeout occurs at the partition within the *timeout* constant time or within the timeout time of another partition with which it has been in contact if this time is later. The timeout occurs at the main partition after it has occurred at the disconnected partitions.
The *timeout* constant is a parameter of the protocol. A timeout is modelled with the timeout action. We start with two simple properties for the *timeout* action.

Property 39 (Members where timeout actions occur) *Timeouts actions only occur on the node that times out, or on a node in the main partition. Formally:*

$$t_i = timeout(p, q) \Rightarrow p = q \vee in\_main\_partition\_at(p, i)$$

Property 40 (Members that can timeout) *Only members outside of the main partition timeout.*

$$t_i = timeout(p, q) \Rightarrow \neg in\_main\_partition\_at(q, i)$$

To determine when a timeout occurs we need to know when a member was split from the main partition.

Definition 22 (Split from main at) *This definition is true if at time t_i, process p is outside of the main partition and the last time it split of from the main partition as a member before t_i was at time t_j.*
Formally:

$$\begin{aligned}
&split_from_main_at(p, j, i) \stackrel{\text{def}}{=} j < i \\
&\wedge partition_of(p, j).main = true \\
&\wedge member_at(p, j) \\
&\wedge \forall k (j < k \leq i \Rightarrow partition_of(t_k, p).main = false)
\end{aligned}$$

We have 2 types of processes:

- For members of the main partition the timeout time is infinite
- For other members it is the highest timeout time of non-main partitions they were a member of until now. The initial timeout time when they split off from the main partition is *timeout* time after that event

We combine these two types in one definition, giving the timeout time of a process p at time t_i.

Definition 23 (Timeout time)

$$timeout\_time\_at(p,i) \stackrel{\text{def}}{=} \begin{cases} \infty & if \quad \exists P(\quad partition\_of(t_i,p) = P \\ & \qquad \wedge P.main = true) \\ & \\ t & if \quad \exists q,j,t(\quad partition\_of(t_j,p) = partition\_of(t_j,q) \\ & \qquad (\exists k(split\_from\_main\_at(p,k,i) \\ & \qquad \ \wedge timeof(t_k) + timeout = t) \\ & \qquad \vee timeout\_time\_at(t_j,q) = t \\ & \qquad) \\ & \quad \wedge\neg\exists q',j',t'(\quad partition\_of(t_{j'},p) = partition\_of(t_{j'},q') \\ & \qquad (\exists k(split\_from\_main\_at(p,k,i) \\ & \qquad \ \wedge timeof(t_k) + timeout = t') \\ & \qquad \vee timeout\_time\_at(t_{j'},q') = t') \\ & \qquad) \\ & \qquad \wedge t < t' \\ & \quad) \end{cases}$$

To enforce the timeouts at the non-main partitions, we simply require that if a certain process' timeout time has expired it will no longer be a member. Whether this has happened because of a timeout event, or because it left or was in a minority partition does not matter.

Property 41 (Proces timeouts)

$$\begin{aligned} & time\_of(t_j) > timeout\_time\_at(p,i) \\ & \quad \Rightarrow \neg is\_member\_at(p,i) \end{aligned}$$

We do something similar for the main partition. First we define shorthands to determine which processes the main partition knows have joined the group and which have left.

Definition 24 (Seen join at main) *Using the seen_join definition, we can define which members the main partition has seen join at any given time.* $seen\_join\_at\_main(p,i)$ *means that at time* t_i *the main partition was aware of the fact that* p *has joined the group.*
Formally:

$$\begin{aligned} seen\_join\_at\_main(p,i) \stackrel{\text{def}}{=} \quad & \exists q,j,k(k \leq i \\ \wedge \quad & seen\_join(q,p,j,k) \\ \wedge \quad & partition\_of(t_k,q).main = true) \end{aligned}$$

Definition 25 (Seen leave at main) *Using the seen_leave definition, we can define which members the main partition has seen join at any given time.* $seen\_leave\_at\_main(p,i)$ *means that at time* t_i *the main partition was aware of the fact that* p *has left the group.*

Formally:

$$
\begin{aligned}
seen\_leave\_at\_main(p,i) \stackrel{\text{def}}{=} \;& \exists q,j,k(k \leq i \\
\wedge \;& seen\_leave(q,p,j,k) \\
\wedge \;& partition\_of(t_k,q).main = true)
\end{aligned}
$$

Next we require that if *all* the processes that the main partition knows to have joined and that are currently not in the main partition have expired timeouts, then the main partition knows about all of them leaving. Whether is knows this because it saw them leave, got this information from another node, or because of a timeout event is not important. Again, if it was not because of the first two, we force a timeout event in this way.

Property 42 (Main partition timeouts)

$$
\begin{aligned}
\forall p \quad & (seen\_join\_at\_main(p,i) \\
& \wedge partition\_of(t_i,p).main = false \\
& \Rightarrow timeof(t_i) > timeout\_time\_at(p,i) \\
&) \\
\Rightarrow \forall p \quad & (seen\_join\_at\_main(p,i) \\
& \wedge partition\_of(t_i,p).main = false \\
& \Rightarrow seen\_leave\_at\_main(p,i)
\end{aligned}
$$

Note that these properties only specify an upper bound after which the timeout must have occurred. At present the specification doesn't prevent the nodes to timeout immediately.

10 Group membership

With these new definitions we can define when a process stops being a member of the group. There are four reasons why a node may be dropped from the group. These four a defined below, and then combined in a new definition for *leave_at*.

Definition 26 (Minority drop) *When a member finds itself in a minority partition, it is no longen a member of the group. Formally:*

$$dropped\_min(p,i) \stackrel{\text{def}}{=} \exists P(minority\_part(p,P,i)$$

Definition 27 (Majority drop) *When a member finds itself in a majority partition, all nodes in the minority partitions are no longer a member of the group. Formally:*

$$\begin{array}{rcc} dropped\_maj(p,i) \stackrel{\text{def}}{=} & \exists q,P,j,P'(majority\_part(q,P,j) \\ \wedge & partition\_of(t_j) = P' \\ \wedge & p \notin P.members \\ \wedge & p \in P'.members \end{array}$$

Definition 28 (Leaving member) *When a member leaves the group, it is no longer a members of the group. Formally:*

$$dropped\_left(p,i) \stackrel{\text{def}}{=} t_i = leave(p)$$

Definition 29 (Timeout drop) *When a member times out, it is no longer a members of the group. Formally:*

$$dropped\_timeout(p,i) \stackrel{\text{def}}{=} t_i = timeout(p,p)$$

Since a member always timeouts at itself first, this will be the first timeout action for p.

Definition 30 (Leave at) *We combine all the different reasons why a node may be dropped, and define this to be the new meaning of leave_at. All the previous definitions still hold with this new definition. Formally:*

$$\begin{array}{rcc} leave\_at(p,i) \stackrel{\text{def}}{=} & dropped\_min(p,i) \\ \vee & dropped\_maj(p,i) \\ \vee & dropped\_left(p,i) \\ \vee & dropped\_timeout(p,i) \end{array}$$

11 Designated joint point

When the last member in a group or a partition leaves, that group or partition would disappear. To prevent this we want the last member to leave the group to become a designated joint point for that group or partition. Nodes can join if they are in contact with at least one member or designated join point. We first define a shorthand to detect when the last member of a partition leaves. We will call this definition $start\_djp\_leave$ because in the next section we will introduce another reason to start a designated join point.

Definition 31 (Start djp leave) *The following shorthand is true if p leaves at t_i and p was the last member in his partition.*

$$\begin{aligned} start\_djp\_leave(p,i) \stackrel{\text{def}}{=} & \quad t_i = leave(p) \\ \wedge & \quad \exists P(P = anticipated\_partition\_of(t_i) \\ \wedge & \quad \neg\exists q(q \in P.members \\ \wedge & \quad member\_at(q,i))) \end{aligned}$$

Definition 32 (Start djp) *A designated join point is started either because the last member in a partition left, or because of a minority recovery, which will be defined in section 12.*
Formally:

$$\begin{aligned} start\_djp\_at(p,i) \stackrel{\text{def}}{=} & \quad start\_djp\_leave(p,i) \\ \vee & \quad start\_djp\_minority\_recovery(p,i) \end{aligned}$$

Next we define when a process stops being a designated join point. This is when

- a process joins at the join point.
- the join point's partition merges with a partition that contains a member.
- the join point's partition merges with a partition that contains another join point.
- the join point times out.

We give definitions for each of these, and then combine then in $stop\_djp$, like we did for $leave\_at$. The definitions for the first two cases are simple.

Definition 33 (Stop djp join)

$$stop\_djp\_join(p,i) \stackrel{\text{def}}{=} \exists q(join\_at(q,p,i))$$

Definition 34 (Stop djp merge member) *When partitions merge and one contains a member and the other a djp, the djp will disappear.*
Formally:

$$\begin{aligned} stop\_djp\_merge\_member(p,i) \stackrel{\text{def}}{=} & \quad \exists P,q(partition(q,P,i) \\ \wedge & \quad p \in P.members \\ \wedge & \quad member\_at(q,i) \end{aligned}$$

When two partitions containing a djp merge, we need a way to decide which djp will stop and which will continue. We will use a function for this, which makes this decision.

Definition 35 (Decide function) *decision(p,q) is a function such that*

$$decision(p,q) = \neg decision(q,p)$$

Definition 36 (Stop djp merge djp) *When two partitions merge that both contain a djp, one will stop being a djp.*
Formally:

$$\begin{array}{rcl} stop\_djp\_merge\_djp(p,i) & \stackrel{\text{def}}{=} & \exists q(partition\_of(t_i,p) = partition\_of(t_i,q) \\ & \wedge & djp\_at(p,i) \wedge djp\_at(q,i) \\ & \wedge & decide(p,q) \end{array}$$

To define when a djp stops because of a timeout we first need to know it's timeout time. This is the highest timeout time of all the processes that it has seen leave since the last anticipated partition. The main partition may have learned about the nodes that left before the last anticipated partition because the nodes that split may have remerged with the main partition. Therefore their timeout times are not safe to use. For the main partitions the djp will never timeout.

Definition 37 (Timeout time of djp)

$$timeout\_of\_djp(p) \stackrel{\text{def}}{=}$$
$$\begin{cases} \infty & \text{if} \quad \exists i(start\_djp\_at(p,i) \wedge partition\_of(t_i).main = true) \\ & \\ t & \text{if} \quad \exists i(start\_djp\_at(p,i) \wedge partition\_of(t_i).main = false) \\ & \quad \wedge \\ & \quad \exists t,j,q(leave\_at(q,j) \\ & \quad \wedge p \in view\_of(t_j).members \\ & \quad \wedge timeout\_time\_of(q,j) = t \\ & \quad \wedge anticipated\_partition\_of(t_j,p) = anticipated\_partition\_of(t_i,p) \\ & \quad \wedge \\ & \quad \neg\exists j',q'(leave\_at(q',j') \\ & \quad \wedge p \in view\_of(t_{j'}).members \\ & \quad \wedge timeout\_time\_of(q,j') > t \\ & \quad \wedge anticipated\_partition\_of(t_{j'},p) = anticipated\_partition\_of(t_i,p))) \end{cases}$$

Definition 38 (Stop djp timeout) *We define a process to stop being a djp at the time of the last action before its timeout has expired.*
Formally:

$$\begin{array}{rcl} stop\_djp\_timeout(p,i) & \stackrel{\text{def}}{=} & time\_of(t_i) < timeout\_of\_djp(p) \\ & \wedge & \neg\exists j(j > i \\ & \wedge & time\_of(t_j) < timeout\_of\_djp(p)) \end{array}$$

Using this we make a general definition for stop djp.

Definition 39 (Stop djp)

$$\begin{array}{rcl} stop\_djp\_at(p,i) & \stackrel{\text{def}}{=} & stop\_djp\_join(p,i) \\ & \vee & stop\_djp\_merge\_member(p,i) \\ & \vee & stop\_djp\_merge\_djp(p,i) \\ & \vee & stop\_djp\_timeout(p,i) \end{array}$$

Definition 40 (djp at)

$$\begin{array}{rcl} djp\_at(p,i) & \stackrel{\text{def}}{=} & \exists j(j < i \wedge start\_djp\_at(p,j) \\ & \wedge & \neg\exists k(j < k < i \wedge stop\_djp\_at(p,k))) \end{array}$$

12 Minority partition recovery

It is possible that because of a partition in three parts, or because of a partition that is caused by a failure, all members find themselves in a minority partition. They will all be dropped from the group, but they may later merge again. When a majority of the members find themselves in the same partition, the can conclude that there was no majority partition. They then recover by forming a designated join point.
The merge rule for the join points makes sure that eventually, there will only be one left. We also want this join point to be aware that all the members in the group were dropped, so that this information may be used to decide the group is unpartitioned.

We start by defining the partition that a member was dropped from because of a minority partition:

Definition 41 (dropped from partition minority) *This definition is true when process p was dropped because of a minority partition, before time t_i, and the last partition before it was dropped is P.*
Formally:

$$\begin{aligned} dropped\_minority\_partition\_before(p,P,i) \stackrel{\text{def}}{=} & \quad \exists j(j < i \\ & \wedge \quad dropped\_min(p,j) \\ & \wedge \quad partition\_of(t_j) = P \\ & \wedge \quad member\_at(t_j)) \end{aligned}$$

A process recovers from a minority partition if it finds more than half of the processes from the partition it was dropped from, and these were all dropped because of a minority, and they were all dropped from the same partition.

Definition 42 (recover from min part)

$$\begin{aligned} & recover\_from\_min\_part(p,i) \stackrel{\text{def}}{=} \\ & \quad \exists P, P' \quad (partition\_at(p,P,i) \\ & \qquad \wedge \quad dropped\_minority\_partition\_before(p,P',i) \\ & \qquad \wedge \quad 2\left|\left\{ \begin{array}{ll} q: & q \in P.processes \wedge \\ & dropped\_minority\_partition\_before(q,P',i) \end{array} \right\}\right| \\ & \qquad > |members\_in(P', end(P'))| \end{aligned}$$

This is the second reason to start being a djp:

Definition 43 (start djp minority recovery)

$$start\_djp\_minority\_recovery(p,i) \stackrel{\text{def}}{=} recover\_from\_min\_part(p,i)$$

Finally we need to define $know\_dropped\_minority(p,q,i)$, which was used in the definition of $seen\_leave(p,q,i)$:

Definition 44 (know dropped minority) *A process that recovers from a minority partition knows that all other members in the partition it was dropped from were also dropped because of a minority partition.*
Formally:

$$\begin{aligned} know\_dropped\_minority(p,q,i) \stackrel{\text{def}}{=} & \quad \exists j,k,l,P(j \le i \wedge k \le i \wedge l \le i \\ \wedge & \quad recover\_from\_min\_part(p,j) \\ \wedge & \quad dropped\_from\_partition\_minority(p,P,k) \\ \wedge & \quad dropped\_from\_partition\_minority(q,P,l)) \end{aligned}$$

References

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[Vit01] Gregory V. Chockler; Idit Keidar; Roman Vitenberg. Group communication specifications: A comprehensive study. *To appear in ACM Computing Surveys*, December 2001.