Membership Algorithms for Multicast Communication Groups

Yair Amir, Danny Dolev*, Shlomo Kramer, Dalia Malki

The Hebrew University of Jerusalem, Israel

Abstract. We introduce a membership protocol that maintains the set of currently connected machines in an asynchronous and dynamic environment. The protocol handles both failures and joining of machines. It operates within a multicast communication sub-system.
It is well known that solving the membership problem in an asynchronous environment when faults may be present is impossible. In order to circumvent this difficulty, our approach rarely extracts from the membership live (but not active) machines unjustfully. The benefit is that our procotol always terminates within a finite time. In addition, if a machine is inadvertently taken out of the membership, it can rejoin it right away using the membership protocol.
Despite the asynchrony, configuration changes are *logically* synchronized with all the regular messages in the system, and appear *virtually synchronous* to the application layer.
The protocol presented here supports partitions and merges. When partitions and merging occur, the protocol provides the application with exact information about the status of the system. It is up to the application designer to merge the partitioned histories correctly.

1 Introduction

We introduce a membership protocol that maintains the set of currently connected machines in an asynchronous and dynamic environment. The protocol handles both failures and joining of machines.

In such an environment, a consistent membership is a key for constructing fault tolerant distributed applications. Machines may have to keep track of other machines in the system. Knowing which machines are connected and active, and even having this knowledge consistent within the set of connected machines can be crucial. The problem of maintaining machine-set membership in the face of machine faults and joins is described in [6].

The protocol presented here is designed to implement the membership maintenance in Transis, a communication sub-system for high availability, currently developed at the Hebrew University of Jerusalem. A Transis *broadcast domain* comprises of a set of machines that can communicate via multicast messages. When sending a message inside this broadcast domain, the Transis sub-system

---

* also at IBM Almaden Research Center

uses the network broadcast capability. Typically, only a single transmission is needed for efficient dissemination of messages to the multiple destinations.

Due to the asynchronous and dynamic properties of the environment, messages can be delayed or can be lost and machines can come up or crash. Moreover, the network itself may partition and re-connect. The *Basic* service of Transis overcomes arbitrary communication delays and message losses and guarantees fast delivery of messages at all of the currently connected destinations. The membership protocol automatically maintains the set of currently connected machines inside the broadcast domain.

The membership protocol is a careful integration of fault and join mechanisms. It preserves several important properties:

- Consensus. It maintains a consistent current configuration among the set of active and connected machines.
- Virtual-synchrony. It guarantees that members of the same configuration receive the same set of messages between every pair of configuration changes.
- Spontaneous. The fault machanism is triggered when a machine detects that communication is broken with another machine for a certain amount of time. The join machanism is triggered when a machine detects a "foreign" message in the broadcast domain. The current set then attempts to merge with the foreign set or sets.
- Symmetric. There are no natural *joining-sides* and *accepting-sides*, and the merging is done multi-way (and not in pairs only).
- Non-blocking. It never blocks indefinitely and it allows regular flow of messages while membership changes are handled.
- Correct handling of partitions and merges. The protocol also handles failures that occur during the join.

The most challenging property of our membership protocol is handling partitions and merges. To the best of our knowledge, all of the previous membership algorithms within similar environments [12, 13, 6, 5, 15, 9] handle the joining of single machines only. However, in reality, when the network includes bridging elements, partitions are likely to occur. In this case, there are two or more sets of machines that need to be joined together. On start-ups, each machine comes up as a singleton-set, and then two or more merge into larger connected sets. Thus, we tackle all aspects of joining: re-connecting partitions, recovery or startup of a single machine, and even moving of a machine from one connected set to another (the latter being a hypothetical scenario, in our view).

Many systems do not allow partitioned execution, for the reason that system consistency might be compromised. We believe that the role of the communication sub-system is to deliver messages where possible, and provide accurate information about the success of message delivery and connection. The application designer should then decide whether execution can continue within the partitions. It is important to emphasize that the complete merging of the partitioned histories is application dependent and therefore is not handled by the membership protocol. The final section of the paper presents applications that can benefit from the support of continuous operation despite partitions.

As noted by others ([8, 7, 11]), solving the membership problem in an asynchronous environment when faults may be present is impossible. There are various approaches for circumventing this difficulty ([6, 15, 5, 13, 12]). Our approach never allows indefinite blocking but rarely extracts from the membership live (but inactive) machines unjustfully. This is the price paid for maintaining a consistent membership within the sets of connected and active machines, in an asynchronous environment, without blocking. In addition, if a machine is inadvertently taken out of the membership, it can rejoin right away using the join mechanism.

Related Work

Early solutions to the membership problem employed synchronous protocols ([6]). The problem with synchronous solutions is that they rely on synchronization properties that are difficult to achieve, and are not supported in standard environments.

The membership algorithm in the Isis system ([15, 5]) operates over reliable communication channels and employs a central coordinator. One of the drawbacks of their algorithm is that during configuration changes the flow of regular messages is suspended until all the previous messages are processed. In contrast, the membership protocol presented here is symmetric and does not disrupt the regular flow of messages.

Later work by Mishra *et al.* ([13]) suggests a distributed membership algorithm, based on causally ordered messages (see exact definition in the next section). In this algorithm, the machines reach eventual agreement on membership changes, but the changes are not coordinated. Our membership protocol extends their work by guaranteeing *virtually synchronous* membership changes at all the machines; in addition, we handle partitions and merges.

The approach taken by Melliar-Smith et al. ([12]) is also distributed, and uses a probabilistic algorithm. The algorithm is based on totally ordered broadcast messages, as supported by the Total algorithm ([11]). The coordinated delivery of totally ordered messages is sufficient for achieving membership consensus, and their algorithm need not send any additional messages. The main shortcoming of algorithms like the membership algorithm based on Total is that with small probability, they might block indefinitely in face of faults ([11, 12]). The protocol presented here differs from [12] in achieving consensus based on causal messages. Our approach never allows indefinite blocking but rarely extracts from the membership live (but not active) machines unjustfully.

2 Transis and the System Model

The system comprises of a set of machines that can dynamically crash and restart, and network(s) that might partition and re-merge. The machines communicate via asynchronous multicast messages. A multicast message leaves its source machine at once to all the machines in the system but may arrive at different times to them. Messages might be lost or delayed arbitrarily, but faults cannot alter messages' contents. Messages are uniquely identified through a pair $<$ sender, counter $>$.

Transis contains the communication layer reponsible for the reliable delivery of messages in the system ([2]). Transis guarantees the *causal* (see [10]) delivery order of messages, defined as the reflexive, transitive closure of:

(1) $m \xrightarrow{cause} m'$ if $\text{receive}_q(m) \rightarrow \text{send}_q(m')$ [2]
(2) $m \xrightarrow{cause} m'$ if $\text{send}_q(m) \rightarrow \text{send}_q(m')$

In Transis, each newly emitted message contains ACKs to previous messages. The ACKs form the $\xrightarrow{cause}$ relation directly, such that if m' contains an ACK to m, then $m \xrightarrow{cause} m'$. If a message arrives at a machine and some of its causal predecessors are missing, Transis transparently handles message recovery and re-ordering. Other environments like [5, 14] are equally suitable for providing the causality requirement. Below, we sometimes refer to the environment and messages as the Transis environment and Transis messages.

The membership protocol operates above the Transis communication layer, such that message arrival order within the protocol preserves causality. We think of the causal order as a directed acyclic graph (DAG): the nodes are the messages, the arcs connect two messages that are directly dependent in the causal order. An example DAG is depicted in Figure 1.

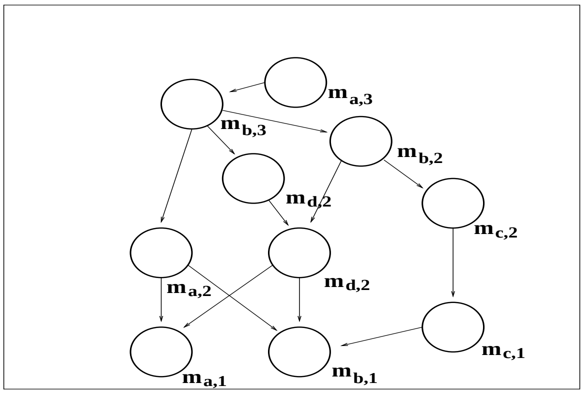

Fig. 1. An Imaginary DAG

The causal graph contains all the messages sent in the system. All the machines eventually see the same DAG, although as they progress, it may be "revealed" to them gradually in different orders. Whenever a message is emitted, it

[2] Note that '$\rightarrow$' orders events occurring at q sequentially, and therefore the order between them is well defined.

causally follows all the messages in the portion of the DAG currently revealed (this results directly from the definition of $\xrightarrow{cause}$).

The Transis communication sub-system receives the messages off the network. It performs recovery and message handling, and at some later time, it *delivers* the messages to the upper level. Transis provides a variety of reliable multicast services. The services use different delivery criteria on the messages in the DAG. In some cases, the membership protocol interferes with the delivery of messages, as we shall see below. The paper [2] provides a detailed description of the Transis environment and services. Here is a short description of the Transis multicast services:

1. **Basic** multicast: guarantees delivery of the message at all the connected sites. This service delivers the message immediately from the DAG to the upper level.
2. **Causal** multicast: guarantees that delivery order preserves causality.
3. **Agreed** multicast: delivers messages in the same order at all sites. The ToTo algorithm implements the agreed multicast service in Transis (see [1]).
4. **Safe** multicast: delivers a message after all the active machines have acknowledged its reception.

The Transis protocols employ the network broadcast capability for the efficient dissemination of messages to multiple destinations via a single transmission.

3 The Membership Problem

The purpose of the membership protocol is to maintain a consistent view of the *current configuration* among all the connected machines in a dynamic environment. This view is used for disseminating reliable multicast messages among all the members. Each machine maintains locally the following view:

CCS: the Current Configuration Set is the set of machines in agreement.

When machines crash or disconnect, the network partitions or re-merges, the connected machines must reconfigure and reach a new agreement on the CCS. The configuration change must take place amidst continuous communication operations. Furthermore, it must indicate to the user which messages are delivered before the reconfiguration and which after. This last property is termed by Birman et al. *virtual synchrony*, and its importance is discussed in [3, 4, 5]. We define the goal of the membership protocol as follows:

P.1 *Maintain the CCS in consensus among the set of machines that are connected throughout the activation of the membership protocol.*

P.2 *Guarantee that any two machines that are connected throughout two consecutive configuration changes deliver the same set of messages between the changes.*

Note that our membership protocol also handles in full multi-way joining. The purpose of the protocol is to merge between two or more membership sets, and to reach an consensus decision on a joined-membership. It is possible however, that only a subset of the live machines succeed in merging their memberships, due to communication delays. This cannot be avoided, since machines might appear silent during the entire joining. Nevertheless, the joined set (or subset) will be in consensus within about its membership.

In order to clarify the discussion and focus on a single execution of the membership protocol, we add to the configuration description the following vector:

Expected: The *Expected* vector contains a message-id per each member of the CCS. This indicates the next message-id from this member following the last configuration change. For example, if before the configuration change, member m has emitted messages up to 19, then $Expected[m] = 20$ [3].

Note that the Expected vector removes unintentional agreement on (recurring) membership sets.

4 Handling Faults

This section focuses on a protocol for handling departure of machines from the set of active ones.

Assume that all the machines belonging to CCS initially agree on the CCS contents. The Faults protocol is initiated every time the communication with any machine breaks. Each machine identifies failures separately. A machine that identifies a *communication-break* with another machine emits a FA message declaring this machine faulty.[4] The FA messages are exchanged within the flow of regular Transis messages and relate to other Transis messages in the regular causal order. The remaining machines in CCS need to agree on the occurred faults. The main difficulty is to concur on the last messages received from crashed machines, because these messages may be delayed arbitrarily long.

Figure 2 depicts a simple scenario of fault handling. In this scenario, machine A lost connection with machine C after message $m_{c,1}$. Consequently, A emits $m_{a,1}$ declaring C faulty. However, machine B has received further messages from C, up to $m_{c,3}$. Therefore, when B concurs via $m_{b,1}$, the causal relations indicate that messages $m_{c,1}$ thru $m_{c,3}$ precede the fault.

[3] In order to provide unique message id's, message id's are pairs (incarnation, counter); Melliar Smith et al. discuss several conditions for providing this uniqueness requirement, among which is the ability to save *incarnation* numbers on nonvolatile storage, see [12]. The *Expected* message-id is therefore either within the current incarnation, or a later one.

[4] The specific method for detecting communication-breaks is implementation dependent and irrelevant to the Faults protocol. For example, in the Transis environment, each machine expects to hear from other machines in the CCS set regularly. Failing this, it attempts to contact the suspected failed machine through a channel reserved for this purpose. If this fails too, it decides that this machine is faulty.

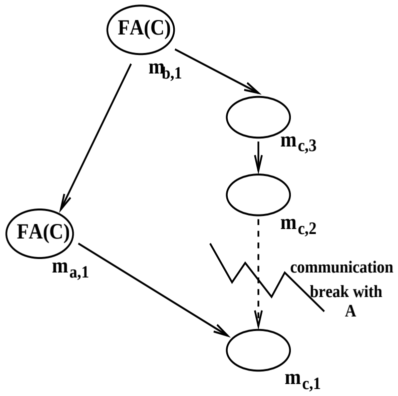

Fig. 2. A Simple Fault Scenario

Generally, this situation is handled as follows: When a FA message is inserted into the DAG, the faults alogrithm marks it *nondeliverable*. It releases FA messages for delivery only after reaching consensus of the *remaining* machines about *all* the faults. Messages of type FA are delivered *last* in their concurrency set (*i.e.* when a FA message causally precedes all the messages in the DAG, it is delivered). For example, in Figure 2, message $m_{a,1}$ is delivered after all of C's messages, since they are all pecedent or concurrent to it. If there are multiple concurrent FA messages, they are delivered in a deterministic order. Note that each fault may be represented by more than one FA message in the DAG; only the first one delivered affects the CCS. In this way, all the machines deliver the same set of causal messages before each configuration change, which guarantees virtual synchrony. The pseudo-code of the protocol is given in Figure 3.

Each iteration of the protocol collects FA messages from other machines (or incurs a communication break with a machine). Within each iteration, a received FA message either increases the F set or results in a consensus decision that shrinks it to $\emptyset$ (in the last step of the protocol). Thus, as the faults protocol is activated many times, F dynamically grows and shrinks; different machines need not assent to the same F set, but eventually they will assent to all the faults contained in the F sets.

Recall that the upper level using Transis is provided with the representation of the current configuration set, the CCS. Initially, CCS in the upper level contains the agreed upon membership set. After a faults set is assented to (at the end of the Faults protocol), the faults are propagated to the upper level via the FA messages in a series of *configuration changes*, and eventually the CCS becomes up-to-date.

As soon as the faults protocol reaches its final step, the internal conditions that require coordination decisions are changed, even before the delivery of the FA messages that incur the configuration change. For every f in the F set, this internal event is called $Crash(f)$. Thus, the protocol assures that the system will not wait indefinitely for failed machines.

Whenever communication breaks with q or a FA message is received:

- if communication breaks with q:
  $f\_set = \{q\}$
- if receive message < FA, f_set > from r:
  LAST[r] = LAST[r] $\cup f\_set$
  mark the FA message non-deliverable
- if ($f\_set \not\subseteq F$)
  $F = F \cup f\_set$
  instruct Transis to disallow any message from f_set to enter[a] *the DAG.*
  broadcast < FA, F >
- if $\forall q \in (CCS \setminus F)$ $LAST[q] = F$
  assent to F
  mark all the FA messages of F *deliverable*
  deliver FA messages last in their concurrency sets
  $F = \emptyset$
  LAST = $\perp$

[a] unless it is already followed by another message in the DAG and required for recovery

Whenever delivering a FA message < FA, f_set > :

$CCS = CCS \setminus f\_set$
$\forall q \in CCS$: set $Expected[q]$ to the message index following the last message delivered from q.

Fig. 3. The Faults Protocol

After delivering a FA message that removes a machine from the configuration, p changes the *Expected* vector: For each machine in the new membership, *Expected* will contain the message id that follows the last delivered message. This id can be either the last counter + 1, or a subsequent incarnation.

4.1 Proof of Correctness

The intuition behind the correctness proof is as follows: Each faults-set F is acknowledged by all the remaining live procesors, before it is accepted and delivered. In the proof we show that this guarantees that the remaining machines achieve consensus about the messages that precede each FA change. If a machine receives a message (m) before a FA message, then the acknowledgement FA it sends follows m in the DAG. Therefore, all the machines that accept FA recover m (if necessary). On the other hand, if m arrives *after* the machine has sent its acknowledgement, m will be discarded, and all the other machines will discard it, too. In this way, all the machines deliver the same messages before each FA configuration change.

First, we introduce some definitions and notations used in the proof. The data structures names are subscribed with the machine id, as in 'DAG_p', in places where it is not obvious from the context.

– A pair of machines p, q are in **membership consent** if

$$CCS_p = CCS_q, \quad \text{and}$$
$$\forall t \in CCS \ : \ Expected_p[t] = Expected_q[t]$$

– A message m from t causally follows the vector $Expected$, denoted $\boldsymbol{Expected} \xrightarrow{cause} \boldsymbol{m}$, if either it is the expected message from t ($Expected[t]$), or $Expected[t] \xrightarrow{cause} m$.
– Denote $\boldsymbol{Accept}_p = CCS_p \setminus F_p$. The $Accept$ set contains the (remaining) machines in CCS that need to assent to F.
– Define:

$$Votes_p(f) = \{m \mid m \in DAG_p, \quad m = <FA, F>, \quad f \in F\}$$
$$Electors_p(f) = \{\text{the first message in } Votes_p(f) \text{ from each sender}\}.$$

The **Electors(f)** set contains the first message from each machine that concurs on f's fault.

Lemma 4.1 *Let p, q be machines currently in membership consent. Assume that p is ready to deliver a configuration change (FA) message CC_f that removes f from the CCS_p. Let $q \in Accept_p$. Then $Electors_q(f) \subseteq Electors_p(f)$.*

Proof: Let $e \in Electors_q(f)$ be a message from e_s. Since p assented to CC_f and since CC_f actually removes f from CCS, DAG_p currently contains FA messages that contain f from all of $Accept_p$. Therefore, if $e_s \in Accept_p$, there is a FA message $e' \in DAG_p$ from e_s that contains f. By the causality property e is also in DAG_p, and by the definition of $Electors_p$, $e \in Electors_p(f)$.

Otherwise, $e_s \in F_p$, where F_p is the faults set that p assented to before delivering CC_f. If any of the machines in $Accept_p$ received the message e before acknoweldging F_p, e will be recovered by p (if necessary), and we are done. Otherwise, all the machines in $Accept_p$ receive e after sending approval to F_p. Therefore, the protocol indicates that they all discard it from the DAG, in contradiction to $e \in Electors_q(f)$. □

If a message follows any of the messages in $Electors_p(f)$, it is delivered only after the configuration change of f. All other messages are delivered before the change. Using Lemma 4.1, we show that the connected machines deliver the same set of causal messages before the configuration change of f.

Lemma 4.2 *Let p, q be machines currently in membership consent. Assume that p and q deliver a configuration change message CC_f that removes f from CCS. Assume that $p \in Accept_q$ and $q \in Accept_p$. Then p and q deliver the same set of causal messages, that follow Expected and precede CC_f.*

Proof: Applying Lemma 4.1 in both directions, we get $Electors_p(f) = Electors_q(f)$. Let m be a message, $Expected \xrightarrow{cause} m$, *s.t.* m is delivered by p before CC_f. Thus, m does not follow any message in $Electors(f)$. If m causally precedes any message in $Electors(f)$, then by the causality property $m \in DAG_q$ and q delivers it before CC_f.

Otherwise, m is concurrent with all the messages in $Electors(f)$. Thus, m was sent by a machine in F_q, F_q being the faults set that q assents to before delivering CC_f. If any machine in $Accept_q$ received m before acknowledging F_q, then q will recover it (if necessary) and deliver it before CC_f. Otherwise, m arrives after all of $Accept_q$ sent their consent to all of F_q, and the protocol indicates that they all discard m.

Similarly, every message delivered by q before CC_f is also delivered by p. □

Theorem 4.3 *Let p, q be machines currently in membership consent. If p, q deliver the configuration change (FA) message CC_f, such that $p \in Accept_q$, $q \in Accept_p$, then:*

1. *p, q deliver the same set of messages following Expected before delivering the configuration change message.*
2. *$Expected_p = Expected_q$ after the delivery.*

Proof: According to Lemma 4.2, the first claim holds. This immediately implies that p and q deliver the same configuration changes following Expected (if any) before CC_f. Therefore, $CCS_p = CCS_q$ after the delivery. Since *Expected* is defined by the CCS and the first item, which we have shown to be equal, the second claim holds. □

The theorem shows that the state of membership consent is preserved after each change. By induction, this holds for every FA message, and the faults (configuration-changes) are delivered in the same order at the machines while preserving the virtual synchrony property.

5 Handling Joins

The join mechanism is trigerred when a machine detects a "foreign" message in the broadcast domain. The current set attempts to merge with the foreign set or sets. Since it operates in a broadcast domain, we expect this to typically happen at the other set(s) and the protocol works symmetrically, *i.e.* there is no joining-side and accepting-side. Note that actual simultaneity is not required for correctness. The closer the sets commence, the sooner they will complete the membership protocol.

Enter **Stage 0** whenever $J = \emptyset$ and intercepting a foreign message.

broadcast < AJ, CCS >
shift to Stage 1

Enter **Stage 1** either from Stage 0, or whenever receiving an AJ message from CCS.

Set an α timer.
$J = CCS$

Whenever receiving an AJ/JOIN message, or a timeout event:

- if receive a message < AJ, j_set > or < JOIN, j_set > from q
  $J = J \cup j\_set$
  if it is a JOIN message then LAST[q] = j_set
- if α expires
  broadcast < JOIN, J >
  shift to Stage 2

Enter **Stage 2** from Stage 1, when α timer has expired.

Whenever receiving a JOIN message:

- if receive message < JOIN, j_set > from $r \in J$
  LAST[r] = j_set
  $J = J \cup j\_set$
  if J changed then broadcast < JOIN, J >
- if $\forall q \in J \;\; LAST[q] = J$
  assent to J
  $CCS = J$
  $J = \emptyset$
  LAST = $\perp$

Fig. 4. The Simplified Join Protocol, no Faults Handling

5.1 The Simplified Join Protocol

As a first step towards the full membership protocol, Figure 4 contains a join protocol for a faultless (asynchronous) environment.

Intuitively, Stage 0 "advertizes" the CCS in an attempt to join. Stage 1 is an optimization step, and its purpose is to collect as many "suggestions" as possible during an α interval. When the α timer expires, the J set gets fixed and Stage 2 starts. In Stage 2, the machine emits a commitment JOIN message. It tries to achieve consensus on J, and messages from machines outside J are ignored. If a member within J emits an expanding suggestion, J is effectively *cancelled.* The principle idea is that in this case, it is **safe** to shift to a different J suggestion, since the old J will never achieve consensus (because a required member will never acknowledge it). In this case, a new commitment is made. Let us first see why the simple faultless join protocol is correct. The following two claims do not constitute a full proof of correctness, and are intended only for demonstration of the main properties of the Simplified Protocol.

Lemma 5.1 *Let p, q be machines in membership consent. If both p and q emit AJ messages, then they emit the same AJ message.*

Proof: AJ messages are emitted only at Stage 0. Therefore, since J is $\emptyset$, there are no pending joinings, and CCS is agreed-on between p and q. $\Box$

We need to show that if p, q are connected, they deliver the same JOIN message. The following assures this:

Lemma 5.2 *Let p be a machine, such that p assents to J_p in the last step of the protocol. Then $\forall q \in J_p$, q assents to J_p.*

Proof: Let $r \in J_p$. Since p assents to J_p, r sent a $JOIN$ message with $J_r = J_p$. Therefore, any previous JOIN suggestion from r is a subset of J_p (J monotonically increases at each machine). Therefore, r did not send a JOIN message cancelling (expanding) J_p up until it sent J_p. Since this is true $\forall q \in J_p$, there are no expanding suggestions from within J_p. But r is committed to J_p after sending it, and considers messages only from within J_p; therefore there is no message cancelling J_p. Since for now we assume no faults and no message losses, eventually, r will receive all the JOIN messages acknowledging J_p. $\Box$

This protocol forms the basis for the full membership protocol. The next step is to handle faults occurring during the joining.

The Complete Membership Protocol

In the Complete Membership Protocol we address the following matters that were left out of the Simplified Join Protocol:

1. Faults handling.
2. Assimilating messages from “foreign” machines during the joining.
3. Preserving the virtual synchrony property.

Fault Handling. The principle idea of consensus decision on faults occurring during the membership protocol is similar to the fautls-handling protocol presented above. The difference is that there are two sets of faults: F_{before} and F_{after} . F_{before} contains fautls that are known before emitting the JOIN messages. These fautls occur effectively in the current membership, before the joining. If there are FA messages concurrent to a JOIN suggestion, a later JOIN suggestion will include them in F_{before} . F_{after} contains faults that are repoirted after the JOIN messages. The join-set, J, and the faults-set, F_{before} , F_{after} , can only *increase* during the protocol, *i.e.* if a machine crashes, it is added to the faults sets and is **not** taken out of J until the joining completes.

Assimilating foreign messages. The symmertical joining relies on the broadcast nature of our environment. The join mechanism is triggered when the machines intercept "foreign" messages. However, the membership protocol requires more than that: it requires reliable and causal message delivery between all the (now) connected machines. Usually this cannot be done unless all the participating machines have already been integrated into the membership [5]. For this purpose, we include a vector with a *cut-counter* for each machine in the joined set. The *cut-vector* is attached to AJ messages. The vector informs the communication system to recover messages for every foreign machine only back to this counter. Messages from machines outside the current membership are kept in a separate DAG called the *completion-DAG* (CDAG) until the membershipmembership completes. These messages cannot be delivered until the joined set is assented to. After the membership protocol terminates, some of these messages appear to belong to some past membership, and are discarded from the CDAG (see below).

Virtual synchrony. Another matter that is not handled by the Simplified Join Protocol in Figure 4 is the preserving of virtual synchrony. The Modified Join Protocol employs special Transis messages (AJ, JOIN, FA), that relate to all other messages in the system according to the regular causal order. The delivery of JOIN messages is delayed until they are assented to or cancelled, in order to guarantee virtually synchronous configuration changes at all the machines.

When the protocol completes, a certain join configuration message $< J, F, >$ is assented to. There are many identical JOIN messages representing $< J, F, >$ in the DAG. The machines update CCS to be $J \setminus F$ upon delivery of the first assented to JOIN message. All the messages in the CDAG that do not follow the assented to JOINs are discarded from the CDAG. The DAG and CDAG are merged, and the joined membership's DAG contains only messages that follow at least one of the assented to JOIN messages.

Note that due to transient communication problems, intersecting sets may be formed. Instead of unifying the join sets of AJ messages, we actually keep a list of joining *sets* (sets are in their original transmitted form). In this way, we can handle the joining of intersecting sets correctly.

The protocol uses the following data structures per machine: J contains the total set of known machines, F_{before} is the set of faulty machines contained in the current JOIN suggestion, F_{after} contains faulty machines that are considered active according to the current JOIN suggestion; J_LAST and F_LAST are arrays containing the most up-to-date JOIN and FA sets respectively received from each machine. We begin by introducing a series of macros in Figure 5. The Complete Membership Protocol is presented in two parts; the purpose of the first part (Figure 6, Stage 0 and 1), is to optimize, by collecting as many foreign join-attempt (AJ) messages, without committing. The second stage (Figure 7) achieves a consensus decision on the join set.

[5] For example, if Transis attempts to recover back messages from any detached set, it might deadlock waiting for messages that have been discarded long ago. The same problem will occur in other environments in similar forms.

BROADCAST-JOIN:

$F_{before} = F_{before} \cup F_{after}$; $F_{after} = \emptyset$
broadcast < JOIN, J, F_{before} >
mark JOIN messages in the DAG that $\neq$ < J, F_{before} > *rejected*

BROADCAST-FA:

broadcast < FA, $F_{before} \cup F_{after}$, J, F_{before} >

INCORPORATE-JOIN < JOIN, j_set, f_set > from r:

$J = J \cup j\_set$; $J\_LAST[r] =$ < j_set, f_set >
$F_{before} = F_{before} \cup f\_set$; $F\_LAST[r] = F\_LAST[r] \cup f\_set$
Mark the JOIN message nondeliverable

INCORPORATE-FA < FA, f_set, f_J, f_F > from r into F_x:

$F_x = F_x \cup f\_set$; $F\_LAST[r] = F\_LAST[r] \cup f\_set$
discard all further messages from F_x

Fig. 5. Macros for the Modified Join Protocol

5.2 Proof of Correctness

We now formulate a series of claims, whose purpose is to guarantee that connected machines do not assent to different JOIN suggestions. The principle idea shown by the following claims is that when a machine "shifts" to a new JOIN message (after committing to a previous one), it is "safe" to do so.

We will refer to the pair < J, F_{before} > as a *suggested join configuration.* We now define *Accept* set to be $J \setminus (F_{before} \cup F_{after})$.

We employ the following two properties of the protocol:

Property 1 During a membership protocol, the J and F_{before} sets at each machine are monotonically increasing.

Property 2 If a machine emits a FA message < FA, f_set, f_J, f_F > , it has already emitted a JOIN message containing < f_J, f_F > .

Lemma 5.3 *If q receives (at Stage 2) a JOIN message < JOIN, j_set, f_set > from $r \in Accept_q$, s.t. $j\_set \not\subseteq J_q$, (similarly $f\_set \not\subseteq F_{before_q}$), then there exists a machine in $Accept_q$ that will either never acknowledge q's current join configuration < J_q, F_{before_q} > , or emit a different join configuration before acknowledging F_{after_q}. Therefore, the join configuration < J_q, F_{before_q} > together with F_{after_q} cannot be assented to.*

Proof: There are two cases: the first, $\exists s \in Accept_q$ that did not acknowledge < J_q, F_{before_q} > and acknowledged j_set (or f_set). By Property 1, s will never acknowledge < J_q, F_{before_q} > , and we are done.

Otherwise, each machine in $Accept_q$ has committed to < J_q, F_{before_q} > at some point. Therefore, in order for any one of them to shift to a different join configuration, it must receive a different suggestion from within $J_q \setminus F_{before_q}$.

Enter **Stage 0** whenever $J = \emptyset$ and $F = \emptyset$ and intercepting a foreign message:

broadcast $<$ AJ, CCS, cut_vector $>$
shift to Stage 1

Enter **Stage 1** either from Stage 0, or when receiving an AJ message from CCS:

Set an α timer.
$J = CCS$.

Whenever receiving an AJ/JOIN message, or a timeout event:

- if receive AJ message $<$ AJ, j_set, cut_vector $>$
  $J = J \cup j\_set$
  extend CDAG recovery for Transis using cut_vector
- if receive JOIN message
  INCORPORATE-JOIN $<$ JOIN, j_set, f_set $>$ from r
- if receive FA message
  INCORPORATE-FA $<$ FA, f_set, F_J, F_f $>$ from r into F_{before}
- if α expires or communication breaks with any machine in CCS
  BROADCAST-JOIN, shift to Stage 2

Fig. 6. The Modified Join Protocol: first stage

Therefore, $\exists s \in Accept_q$ that receives a different join suggestion from F_{after_q}. Machine s receives this messages before acknowledging F_{after_q} (otherwise it discards the message). Thus, before s acknowledges F_{after_q} it shifts to a new join configuration. Therefore, since the communication is FIFO between s and q, s never acknowledges both J_q, F_{before_q} and F_{after_q}. $\Box$

Lemma 5.4 *If q receives (at Stage 2) a FA message $<$ FA, f_set, f_J, f_F $>$ from $r \in Accept_q$, s.t.*

(1) $J_p \neq f_J$ *or* $F_{before_p} \neq f_F$, *and*
(2) $f\_set \not\subseteq F_{before_q}$

Then r will never acknowledge q's current join configuration $< J_q, F_{before_q} >$.

Proof: Since the communication between r and q is FIFO, and Property 2 holds at r, then $f_J \subset J_q$ and $f_F \subset F_{before_q}$ (at least one inclusion is strong). Therefore, Property 1 assures that r did not acknowledge q's join configuration before this FA message. Any following JOIN message from r will contain f_set in its F_{before} field, and will differ from F_{before_q}. $\Box$

Lemma 5.3 and Lemma 5.4 cover all the cases in which the protocol indicates to shift to a new JOIN message (emit another JOIN message). The correctness claim is a direct consequence of this.

Enter **Stage 2** from Stage 1, when α timer has expired.

Whenever receiving a JOIN/FA message:

- if receive JOIN message from $q \in J \setminus (F_{before} \cup F_{after})$
  - if $j\_set \not\subseteq J$ or $f\_set \not\subseteq F_{before}$
    - INCORPORATE-JOIN < JOIN, j_set, f_set > from q
    - BROADCAST-JOIN
  - else
    - INCORPORATE-JOIN < JOIN, j_set, f_set > from q
- if receive FA message from $q \in J \setminus (F_{before} \cup F_{after})$
  - if $f_J = J$ and $f_F = F_{before}$
    - INCORPORATE-FA < FA, f_set, f_J, f_F > into F_{after}
    - BROADCAST-FA
  - else if $f\_set \not\subseteq F_{before}$
    - INCORPORATE-FA < FA, f_set, f_J, f_F > into F_{before}
    - BROADCAST-JOIN
  - else $F\_LAST[q] = F\_LAST[q] \cup f\_set$
- if communication breaks with $q \in J \setminus (F_{before} \cup F_{after})$
  - $F_{after} = F_{after} \cup \{q\}$
  - BROADCAST-FA
- let $F = F_{before} \cap CCS$
  if $\forall q \in (CCS \setminus F) \;\; F\_LAST[q] \supseteq F$
  - *assent to* F
  - *mark all FA messages in* F *deliverable*
- if $\forall q \in J \setminus (F_{before} \cup F_{after})$:
  $J\_LAST[q] =$ < J, F_{before} > and $F\_LAST[q] \supseteq F_{after}$
  - *assent to* < J, F_{before}, F_{after} >
  - *mark all JOIN messages* < J, F_{before} > *deliverable*
  - *mark all FA messages in* F_{after} *deliverable*
  - merge DAG and CDAG from the join point

Fig. 7. The Modified Join Protocol: second stage

Whenever delivering a JOIN message $j =$ < f_set, f_b, f_a > :

$CCS = j\_set \setminus f_b$
$\forall q \in CCS$: set $Expected[q]$ to the message index of q's message in Electors(j).
if there is none, set $Expected[q]$ to the message index following the last message delivered from q.

Fig. 8. The Modified Join Protocol: delivery

Theorem 5.5 *Let p be a machine. If p assents to $J_p \setminus (F_{before_p} \cup F_{after_p})$ (in the last step), and $q \in Accept_p$, then q cannot assent to any different join configuration, other than $< J_p,\ F_{before_p} >$.*

Proof: Since $q \in Accept_p$, there is a point in q's execution when it acknowledged $< J_p,\ F_{before_p} >$ and F_{after_p}, *i.e.* it emitted a FA message $<$ FA, $F_{before_p} \cup F_{after_p},\ J_p,\ F_{before_p} >$. Denote with **Situation 0** the situation comprising of q's state at this point, and of p's state when it assents to $J_p \setminus (F_{before_p} \cup F_{after_p})$. At Situation 0, $F_{before_p} = F_{before_q}$, $J_p = J_q$ and $Accept_p = Accept_q$. Machine q is committed to this configuration at Situation 0. Trivially, q could not have assented to any previous configuration, and did not emit any expanding JOIN message before Situation 0. Therefore, we need to prove that given that p has assented to this configuration, q will not shift to a different join configuration after situation 0. There are two possible scenarios in which q might have shifted to a new join configuration:

- If q received a JOIN message $<$ JOIN, j_set, f_set $>$ from $r \in Accept_q$, s.t. $j\_set \not\subseteq J_q$ or $f\_set \not\subseteq F_{before_q}$.
- If q received a FA message $<$ FA, f_set, f_J, f_F $>$ from $r \in Accept_q$, s.t. $J_q \neq f_J$ or $F_{before_q} \neq f_F$, and $f\_set \not\subseteq F_{before_q}$.

In the two cases, lemmata 5.3, 5.4 show that there exists a machine in $Accept_q = Accept_p$ that never acknowledges this join configuration (including F_{after_p}), in contradiction to the fact that p assented to it. □

5.3 Proof of Virtual Synchrony Property

The configuration changes that represent joining are required to preserve **P.2**, the virtual synchrony property for the higher level applications. Recall that the joining change is assented to when all the required members (excluding the faults) send identical JOIN messages . We define:

$$Electors_p = \{m \mid\ m = < JOIN, J_p, F_{before_p} > ,\ m \in DAG_p\}.$$

Electors is the set of identical JOIN messages that promote the assentation to $< J_p,\ F_{before_p} >$. The first JOIN message delivered from $Electors_p$ makes the join configuration change. The remaining JOIN messages are discarded as soon as they become deliverable. Before making the join change, p delivers all the messages prior or concurrent with $Electors_p$. The merged DAG after this joining contains all the messages that follow **any one** of $Electors_p$. This is defined exactly by the *Expected* vector after the joining, $\forall q \in CCS_p:\ \ Expected_p[q] = next(e_q)$, where $e_q \in Electors_p$ is the JOIN message from q. The merged DAG contains the messages that follow $Expected_p$.

We already proved that the join configuration $< J_p,\ F_{before_p} >$ is assented by all of $J_p \setminus (F_{before_p} \cup F_{after_p})$. We proceed to show they have identical *Electors* sets.

Lemma 5.6 *Let p be a machine. Every JOIN message emitted by any machine in J_p follows the cut-vector known to p.*

Proof: This results from the fact that each machine either sends an AJ message, or shifts to Stage 1 when it receives any JOIN or AJ message within its CCS. Thus, all the JOIN messages from each set follow all the emitted cut-vectors. □

Lemma 5.7 *Let p be a machine. If p assents to $< J_p, F_{before_p} >$ (and F_{after_p}), s.t. $q \in Accept_p$, then $Electors_q \subseteq Electors_p$.*

Proof: Let $j \in Electors_q$. If the sender of j is in $Accept_p$, then surely p waits for this message before delivering the join configuration change. Therefore, using Lemma 5.6 $j \in Electors_p$. Otherwise, j's sender is in F_{after_p}. If any machine in $Accept_p$ received j before acknowledging F_{after_p}, then p will recover j (if necessary) and have $j \in Electors_p$. Otherwise, all the machines in $Accept_p$ acknowledge all of F_{after_p} before receiving j, and they all agree to discard it from the DAG, in contradiction to $j \in Electors_q$. □

Theorem 5.8 *Let p, q be machines. Assume that p, q deliver their next JOIN configuration change message CC_p, CC_q respectively, s.t. $p \in Accept_q$ and $q \in Accept_p$. Then:*

1. *$CC_p = CC_q$.*
2. *p and q agree on Expected after the delivery.*

Proof: The first claim follows directly from Theorem 5.5. Therefore, p and q have the same CCS set after delivering CC. From Lemma 5.7, we have $Electors_p = Electors_q$, which further shows that the Expected array for the CCS is the same in p and q. □

Thus, we have shown in this theorem that every two machines that merge in a join procedure, reach membership consent.

Theorem 5.9 *Let p, q be machines in membership consent. Assume that p, q deliver their next configuration change message CC_p, CC_q respectively, s.t. $p \in Accept_q$ and $q \in Accept_p$.*

1. *p, q deliver the same set of messages following Expected before delivering the CC message.*
2. *$CC_p = CC_q$*
3. *$Expected_p = Expected_q$ after the delivery.*

Proof: The proof of the first claim is identical to the proof of Theorem 4.3, replacing FA messages with general configuration-changes. We do not repeat it.

Claims 2 and 3 are contained in Theorem 5.8. □

Thus, we have shown that if p, q are in membership consent and remain connected, then they deliver the same configuration changes and remain in consent. Furthermore, they deliver the same set of messages between configuration changes.

5.4 Proof of Liveness

The membership protocol is provably live if the following two assumptions hold:

1. The set of machines reachable in the system is finite.
2. The system produces *admissible histories* defined as follows: For each message m and each machine p, within a finite time there is either a message from p following m or p is *extracted* by a FA message declaring it faulty.

Lemma 5.10 *Assume p is in Stage 2 of the protocol. Then within a finite time, p either assents to the join configuration $J_p \setminus (F_{before_p} \cup F_{after_p})$, or one of the sets J_p, F_{before_p}, F_{after_p} increases.*

Proof: Let $m_j \in DAG_p$ be a JOIN or FA message containing $< J_p, F_{before} >$, F_{after_p}. According to the assumptions, p receives a message referring to m_j from every machine in $Accept_p$ within a finite time, or a machine is declared faulty. There are a few possibilities:

1. If any message from $Accept_p$ expands the suggestion, either J_p or F_{before_p} increases, as required.
2. If there is any FA message following m_j that contains faults from $Accept_p$, then F_{after_p} increases.
3. Otherwise, the situation is that there are messages referring to m_j from all of $Accept_p$, *s.t.* none of them extends the configuration or suggests any new faults; therefore, the configuration can be assented to. □

Theorem 5.11 *Let p be a machine that starts the membership protocol. Then p completes the protocol within a finite time.*

Proof: Machine p moves to Stage 2 in a constant α delay. In Stage 2, according to Lemma 5.10, p either accepts its current configuration within a finite time, or increases one of the sets J, F_{before} , F_{after} . Since by the assumptions, this growths is limited, this can occur a finite number of times. □

In our implementation we use a strong extraction rule, using timeout for extraction. In this case, the liveness claim may have a definite time bound for completion. However, for purposes of the proof, it is sufficient to assume *eventual* extraction, and show eventual termination accordingly.

6 Discussion

The maintenance of dynamic membership in a distributed environment is essential for the construction of distributed fault tolerant applications. We exemplify this through the following list of applications.

- A general *consensus object* may be implemented over the dynamic membership in a deadlock free manner. The method is straight forward: each member machine sends a 'value' in a message. The decision value is any deterministic function of the collected values. If any of the machines should fail during the procedure, the remaining members learn about the failure within a finite time and proceed to make the decision using the subset of the values. Note that this subset is the same at all the machines, due to the virtual synchrony property. A more complicated consensus decision that utilizes the dynamic membership is given in [1].
- Fault tolerant mutual exclusion can be achieved. If the holder of a lock should fail, the remaining machines can retrieve it.
- A set of coordinated processes can provide reliable work-sharing. In this application, a certain set of tasks is distributed among replicated processes, each performing a certain portion. If one of the worker-processes should fail, the remaining machines can reclaim the portion of the work assigned to it. Note that it is imperative to have up-to-date information about the state of the failed worker in order to know which interactions (*e.g.* with clients) were completed before the failure.

Transis is a transport layer that supports partitioned operation, using the membership protocol described above. For example, assume there are 50 workstations in the computer science department that execute a distributed application. If the network is partitioned into two halves, such that each half contains exactly 25 workstations, each half will gradually remove all the machines in the other half out of its membership, and continue operation normally. When the network reconnects, the membership protocol will merge the partitions, providing the upper level with the exact point in the processing when the join occurs. It is up to the high level application designer to implement a consistent joining.

We give a few examples of applications that may benefit from the ability to operate in partitions:

- A network of ATMs that exhibits partitions should allow some transactions on tellers that are disconnected from the main computer. For example, each partition of tellers can answer to queries on balance and credit and provide the most recent information present in the partition. A partition can allow small amounts to be withdrawn in some cases.
- An airline reservation system can have a standard scheme for dividing the available tickets between partitions. When a partition occur, each partition takes a fixed pre-agreed portion of the available tickets and handles them (perhaps allowing a margin of 10% to remain free, just in case).

The applications listed above must handle re-merging carefully, in an application dependent manner. The guarantee of virtual synchrony by Transis facilitates this merging.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Total ordering of messages in broadcast domains. Technical Report CS92-9, Dept. of Comp. Sci., the Hebrew University of Jerusalem, 1992.
2. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *FTCS conference*, number 22, pages 76–84, July 1992. previous version available as TR CS91-13, Dept. of Comp. Sci., the Hebrew University of Jerusalem.
3. K. Birman, R. Cooper, and B. Gleeson. Programming with process groups: Group and multicast semantics. TR 91-1185, dept. of Computer Science, Cornell University, Jan 1991.
4. K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Ann. Symp. Operating Systems Principles*, number 11, pages 123–138. ACM, Nov 87.
5. K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. TR 91-1192, dept. of comp. sci., Conrell University, 91. revised version of 'fast causal multicast'.
6. F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Research Report RJ 5964, IBM Almaden Research Center, Mar. 1988.
7. D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.
8. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.
9. A. Griefer and R. Strong. Dcf: Distributed communication with fault tolerance. In *Ann. Symp. Principles of Distributed Computing*, number 7, pages 18–27, August 1988.
10. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 78.
11. P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. Parallel & Distributed Syst.*, (1), Jan 1990.
12. P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *Intl. Conf. Distributed Computing Systems*, May 91.
13. S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In *proc. of the intl. working conf. on Dependable Computing for Critical Applications*, Feb 1991.
14. L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 89.
15. A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. TR 91-1188, Dept. of Computer Science, Cornell University, Feb 1991.