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 unjustly. The benefit is that our protocol always terminates within a finite time. In addition, if a machine is inadvertently taken out of the membership, it can join 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 be lost and machines can come up or crash. Moreover, the network itself may partition and re-connect. The 

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 mechanism is triggered when a machine detects that communication is broken with another machine for a certain amount of time. The join mechanism 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 unjustly. 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 unjustly.

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 responsible 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. \( m \xrightarrow{\text{receive}} m' \) if \( \text{receive}_i(m) \rightarrow \text{send}_j(m') \)
2. \( m \xrightarrow{\text{send}} m' \) if \( \text{send}_i(m) \rightarrow \text{send}_j(m') \)

In Transis, each newly emitted message contains ACKs to previous messages. The ACKs form the \( \xrightarrow{\text{send}} \) relation directly, such that if \( m' \) contains an ACK to \( m \), then \( m \xrightarrow{\text{send}} m' \). 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](image)

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 \( \rightarrow^{\text{send}} \) 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 \( \text{causal} \)).

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 reconfiguration 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 $\text{Expected}[m] = 20$.

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. The FA messages are exchanged within the flow of regular Transit messages and relate to other Transit messages in the regular causal order. The remaining machines in CCS need to agree on the occurred faults, 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 $m_{3,1}$. Consequently, A emits $m_{3,1}$ declaring C faulty. However, machine B has received further messages from C, up to $m_{3,5}$. Therefore, when B concurs via $m_{3,3}$, the causal relations indicate that messages $m_{3,4}$ thru $m_{3,5}$ 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 Transit 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.
Generally, this situation is handled as follows: When a FA message is inserted into the DAG, the Fault Algorithm marks it \textit{nondeliverable}. It releases FA messages for delivery only after reaching consensus of the \textit{remaining} machines about \textit{all} the faults. Messages of type FA are delivered \textit{last} in their concurrency set (\textit{i.e.,} when a FA message causally precedes all the messages in the DAG, it is delivered). For example, in Figure 2, message $m_{0,1}$ is delivered after all of $C$'s messages, since they are all precedent 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 assign to the same $F$ set, but eventually they will assign 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 assigned to (at the end of the Faults protocol), the faults are propagated to the upper level via the FA messages in a series of \textit{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 $\textit{Crash}(f)$. Thus, the protocol assumes 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 $q$:
  
  $f_{\text{set}} = \{q\}$

- if receive message $<\text{FA}, f_{\text{set}}>$ from $r$:
  
  \[ \text{LAST}[r] = \text{LAST}[r] \cup f_{\text{set}} \]

  mark the FA message non-deliverable

- if $(f_{\text{set}} \not\subseteq F)$

  \[ F = F \cup f_{\text{set}} \]

  instruct Transis to disallow any message from $f_{\text{set}}$ to enter the DAG.

  broadcast $<\text{FA}, F>$

- if $\forall q \in (\text{CCS} \setminus F)$  \[ \text{LAST}[q] = F \]

  assent to $F$

  mark all the FA messages of $F$ deliverable

  deliver FA messages last in their concurrency sets

  $F = \emptyset$

  $\text{LAST} = \perp$

\( \text{\footnotesize as unless it is already followed by another message in the DAG and required for recovery} \)

Whenever delivering a FA message $<\text{FA}, f_{\text{set}}>$:

\[ \text{CCS} = \text{CCS} \setminus f_{\text{set}} \]

\( \forall q \in \text{CCS} : \text{set Expected}[q] \) to the message index following the last message delivered from $q$.

\[ \text{\footnotesize as can be either the last counter + 1, or a subsequent incarnation} \]

Fig. 3. The Faults Protocol

After delivering a FA message that removes a machine from the configuration, $p$ changes the $\text{Expected}$ vector: For each machine in the new membership, $\text{Expected}$ 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 processors, 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 $\text{DAG}_p$, in
places where it is not obvious from the context.

- A pair of machines $p, q$ are in **membership consent** if
  \[ CCS_p = CCS_q, \quad \text{and} \quad \forall t \in CCS : \text{Expected}_p[t] = \text{Expected}_q[t] \]

- A message $m$ from $t$ causally follows the vector Expected, denoted \text{Expected} \to m, if either it is the expected message from $t$ (Expected[t]), or Expected[t] \to m.

- Denote \text{Accept}_p = CCS_p \setminus F_p. The Accept set contains the (remaining) machines in CCS that need to assert to $F$.

- Define:
  \[
  \text{Votes}_p(f) = \{m \mid m \in \text{DAC}_p, \ m = \langle FA, F \rangle, \ f \in F \}
  \]
  \[
  \text{Elector}_p(f) = \{\text{the first message in } \text{Votes}_p(f) \text{ from each sender}\}.
  \]

The Elector$(f)$ 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 $q \in \text{Accept}_p$. Then Elector$_q(f) \subseteq$ Elector$_p(f)$.

**Proof:** Let $e \in$ Elector$_q(f)$ be a message from $e_p$. Since $p$ asserted to $CC_f$ and since $CC_f$ actually removes $f$ from CCS, $\text{DAC}_p$ currently contains FA messages that contain $f$ from all of $\text{Accept}_p$. Therefore, if $e_p \in \text{Accept}_p$, there is a FA message $e' \in \text{DAC}_p$ from $e_p$ that contains $f$. By the causality property $e$ is also in $\text{DAC}_p$, and by the definition of Elector$_p$, $e \in$ Elector$_p(f)$.

Otherwise, $e_p \in F_p$, where $F_p$ is the faults set that $p$ asserted to before delivering $CC_f$. If any of the machines in $\text{Accept}_p$ received the message $e$ before acknowledging $F_p$, $e$ will be recovered by $p$ (if necessary), and we are done. Otherwise, all the machines in $\text{Accept}_p$ receive $e$ after sending approval to $F_p$. Therefore, the protocol indicates that they all discard it from the DAG, in contradiction to $e \in$ Elector$_q(f)$. $\square$

If a message follows any of the messages in Elector$_q(f)$, 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 $p \in \text{Accept}_q$ and $q \in \text{Accept}_p$. 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 $\text{Electors}_p(f) = \text{Electors}_q(f)$. Let $m$ be a message, $Expected \xrightarrow{\tau} m$, s.t. $m$ is delivered by $p$ before $CC_f$. Thus, $m$ does not follow any message in $\text{Electors}(f)$. If $m$ causally precedes any message in $\text{Electors}(f)$, then by the causality property $m \in DAC_{q}$ and $q$ delivers it before $CC_f$.

Otherwise, $m$ is concurrent with all the messages in $\text{Electors}(f)$. 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 $\text{Accept}_q$ 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 $\text{Accept}_q$ 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 $p$. \(\square\)

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 $p \in \text{Accept}_q$, $q \in \text{Accept}_p$, 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. \(\square\)

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 triggered 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 \( < A, CCS > \)
- shift to Stage 1

Enter Stage 1 either from Stage 0, or whenever receiving an \( A \cdot J \) message from CCS.

- Set an \( \alpha \) timer.
  \[ J = CCS \]
- Whenever receiving an \( A \cdot J \) JOIN message, or a timeout event:
  - if receive a message \( < A, j \cdot set > \) or \( < \text{JOIN}, j \cdot set > \) from \( q \)
    \[ J = J \cup j \cdot set \]
    if it is a JOIN message then \( \text{LAST}[q] = j \cdot set \)
  - if \( \alpha \) expires
    - broadcast \( < \text{JOIN}, J > \)
    - shift to Stage 2

Enter Stage 2 from Stage 1, when \( \alpha \) timer has expired.

Whenever receiving a JOIN message:

- if receive message \( < \text{JOIN}, j \cdot set > \) from \( r \in J \)
  \[ \text{LAST}[r] = j \cdot set \]
  \[ J = J \cup j \cdot set \]
  if \( J \) changed then broadcast \( < \text{JOIN}, J > \)
- if \( \forall q \in J \), \( \text{LAST}[q] = J \)
  - \( \text{assent to } J \)
  \[ CCS = J \]
  \[ J = \emptyset \]
  \[ \text{LAST} = \perp \]

**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 "advertises" 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 \( \alpha \) interval. When the \( \alpha \) 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 *amended*. 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 \( AI \) messages, then they emit the same \( AI \) message.

**Proof:** \( AI \) messages are emitted only at Stage 0. Therefore, since \( J = \emptyset \), 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 \( p \) asserts to \( J_p \) in the last step of the protocol. Then \( \forall q \in J_p, q \) asserts to \( J_p \).

**Proof:** Let \( r \in J_p \). Since \( p \) asserts 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 faults-handling protocol presented above. The difference is that there are two sets of faults: \( F_{before} \) and \( F_{after} \). \( F_{before} \) contains faults that are known before emitting the JOIN messages. These faults 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 reported 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 symmetrical 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. For this purpose, we include a vector with a count for each machine in the joined set. The count 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 membership membership completes. These messages cannot be delivered until the joined set is asserted 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 asserted to or cancelled, in order to guarantee virtually synchronous configuration changes at all the machines.

When the protocol completes, a certain join configuration message \( \langle J, F, > \) is asserted to. There are many identical JOIN messages representing \( \langle J, F, > \) in the DAG. The machines update CCS to be \( J \setminus F \) upon delivery of the first asserted to JOIN message. All the messages in the CDAG that do not follow the asserted 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 asserted 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.

\[ \text{For example, if Transis attempts to recover back messages from any detached set, it might deadlock waiting for messages that have been discarded long ago. The same problem will occur in other environments in similar forms.} \]
**5.2 Proof of Correctness**

We now formulate a series of claims, whose purpose is to guarantee that connected machines do not asent 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 \( \text{Accept}_q \) 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 \( < FA, f_{set}, f_J, f_R > \), it has already emitted a JOIN message containing \( < f_J, f_R > \).

**Lemma 5.3** If \( q \) receives (at Stage 2) a JOIN message \( < JOIN, j_{set}, f_{set} > \) from \( r \in \text{Accept}_q, s.t. j_{set} \not\subseteq J_q \) (similarly \( f_{set} \not\subseteq F_{before} \)), then there exists a machine in \( \text{Accept}_q \) that will either never acknowledge \( q \)'s current join configuration \( < J_q, F_{before} > \), or emit a different join configuration before acknowledging \( F_{after} \). Therefore, the join configuration \( < J_q, F_{before} > \) together with \( F_{after} \) cannot be assented to.

**Proof:** There are two cases: the first, \( \exists s \in \text{Accept}_q \) that did not acknowledge \( < J_q, F_{before} > \) and acknowledged \( j_{set} \) (or \( f_{set} \)). By Property 1, \( s \) will never acknowledge \( < J_q, F_{before} > \), and we are done.

Otherwise, each machine in \( \text{Accept}_q \) has committed to \( < J_q, F_{before} > \) 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 \( J_q \setminus F_{before} \).
Enter Stage 0 whenever \( J = \emptyset \) and \( F = \emptyset \) and intercepting a foreign message:
- broadcast \(< A_1, CC_1, cut\_vector >\)
- shift to Stage 1

Enter Stage 1 either from Stage 0, or when receiving an \( AJ \) message from \( CC_1 \):
- Set an omega timer,
  \( J = CC_1 \).
Whenever receiving an \( AJ/JOIN \) message, or a timeout event:
- if receive \( AJ \) message \(< A_1, j\_set, cut\_vector >\)
  \( J = J \cup j\_set \)
  extend CBAG 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 \( \alpha \) expires or communication breaks with any machine in \( CC_1 \)
  BROADCAST-JOIN, shift to Stage 2

**Fig. 6.** The Modified Join Protocol: first stage

Therefore, \( \exists s \in Accept_1 \) that receives a different join suggestion from \( F\_after_1 \). Machine \( s \) receives this messages before acknowledging \( F\_after_1 \) (otherwise it discards the message). Thus, before \( s \) acknowledges \( F\_after_1 \) it shifts to a new join configuration. Therefore, since the communication is FIFO between \( s \) and \( q \), \( s \) never acknowledges both \( J_1, F\_before_1 \) and \( F\_after_1 \). 

**Lemma 5.4** If \( q \) receives (at Stage 2) a FA message \(< FA, f\_set, f_j, f_F >\) from \( r \in Accept_1 \), s.t.

1. \( J_0 \neq f_j \lor F\_before_1 \neq f_F \), and
2. \( f\_set \supseteq F\_before_1 \)

Then \( r \) will never acknowledge \( q \)’s current join configuration \(< J_1, F\_before_1 >\).

**Proof:** Since the communication between \( r \) and \( q \) is FIFO, and Property 2 holds at \( r \), then \( f_j \subseteq J_0 \) and \( f_F \subseteq F\_before_1 \) (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 \( F\_before \). 

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 a timer has expired.

Whenever receiving a JOIN/FA message:
- if receive JOIN message from \( q \in J \setminus (F_{before} \cup F_{after}) \)
  
  \( \text{if } j_{set} \not\subseteq J \text{ or } f_{set} \not\subseteq F_{before} \)
  
  \( \text{INCORPORATE-JOIN } < \text{JOIN}, j_{set}, f_{set} > \text{ from } q \)
  
  \( \text{BROADCAST-JOIN} \)
  
  \( \text{else} \)
  
  \( \text{INCORPORATE-JOIN } < \text{JOIN}, j_{set}, f_{set} > \text{ from } q \)
  
  \( \text{if } f_{sr} = J \text{ and } f_{fr} = F_{before} \)
  
  \( \text{INCORPORATE-FA } < \text{FA}, f_{set}, f_{sr}, f_{fr} > \text{ into } F_{after} \)
  
  \( \text{BROADCAST-FA} \)
  
  \( \text{else if } f_{set} \subseteq F_{before} \)
  
  \( \text{INCORPORATE-FA } < \text{FA}, f_{set}, f_{sr}, f_{fr} > \text{ into } F_{before} \)
  
  \( \text{BROADCAST-FA} \)
  
  \( \text{else } F_{LAST}[q] = F_{LAST}[q] \cup f_{set} \)
  
- if communication breaks with \( q \in J \setminus (F_{before} \cup F_{after}) \)
  
  \( F_{after} = F_{after} \cup \{ q \} \)
  
  \( \text{BROADCAST-FA} \)
  
- let \( F = F_{before} \cap CCS \)
  
  \( \text{if } \forall q \in (CCS \setminus F) \quad F_{LAST}[q] \supseteq F \)
  
  \( \text{assent to } F \)
  
  \( \text{mark all FA messages in } F \text{ deliverable} \)

- if \( \forall q \in J \setminus (F_{before} \cup F_{after}) : \)
  
  \( F_{LAST}[q] = \langle J, F_{before} > \text{ and } F_{LAST}[q] \supseteq F_{after} \)
  
  \( \text{assent to } < J, F_{before}, F_{after} > \)
  
  \( \text{mark all JOIN messages } < J, F_{before} > \text{ deliverable} \)
  
  \( \text{mark all FA messages in } F_{after} \text{ deliverable} \)
  
  \( \text{merge DAG and CDAG from the join point} \)

**Fig. 7.** The Modified Join Protocol; second stage

Whenever delivering a JOIN message \( j = < f_{set}, f_{sr}, f_{fr} > : \)

\( CCS = j_{set} \setminus f_{fr} \)

\( \forall q \in CCS : \text{set } Expected[q] \text{ to the message index of } q \text{'s message in } Electors(q) \).

\( \text{if there is none, set } Expected[q] \text{ to the message index following the last message delivered from } q. \)

**Fig. 8.** The Modified Join Protocol; delivery
Theorem 5.5. Let p be a machine. If p asserts to $J_p \setminus (F_{before_p} \cup F_{after_p})$ (in the last step), and $q \in \text{Accept}_p$, then q cannot assert to any different join configuration, other than $< J_p, F_{before_p} >$.

Proof: Since $q \in \text{Accept}_p$, 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 $< FA, F_{before} \cup F_{after_p}, J_p, F_{before_p}>$. Denote with Situation 0 the situation comprising of q’s state at this point, and of p’s state when it asserts 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 $\text{Accept}_q = \text{Accept}_p$. Machine q is committed to this configuration at Situation 0. Trivially, q could not have asserted 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 asserted 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 $< \text{JOIN}, j_{set}, f_{set}>$ from r $\in \text{Accept}_q$, s.t. $j_{set} \not\subseteq J_q$ or $f_{set} \not\subseteq F_{before_q}$.
- If q received a FA message $< FA, f_{set}, j_q, f_r>$ from r $\in \text{Accept}_q$, s.t. $J_q \neq J_r$ or $F_{before_q} \neq F_r$, and $f_{set} \not\subseteq F_{before_q}$.

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

5.3 Proof of Virtual Synchrony Property

The configuration changes that represent joining are required to preserve P2, the virtual synchrony property for the higher level applications. Recall that the joining change is asserted to when all the required members (excluding the faults) send identical JOIN messages. We define:

$$\text{Electors}_p = \{ m \mid m = < \text{JOIN}, J_p, F_{before_p} >, m \in \text{DAG}_p \}.$$

$\text{Electors}_p$ is the set of identical JOIN messages that promote the assertion to $< J_p, F_{before_p} >$. The first JOIN message delivered from $\text{Electors}_p$ 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 $\text{Electors}_p$. The merged DAG after this joining contains all the messages that follow any one of $\text{Electors}_p$. This is defined exactly by the Expected vector after the joining, $\forall q \in \text{CSP}_p: \text{Expected}_q[q] = \text{next}(e_q)$, where $e_q \in \text{Electors}_p$ is the JOIN message from q. The merged DAG contains the messages that follow $\text{Expected}_q$.

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

**Lemma 5.7** Let \( p \) be a machine. If \( p \) asserts to \( J_p, F_{before} > (F_{after}) \) and \( q \in \text{Accept}_p \), then \( \text{Electors}_q \subseteq \text{Electors}_p \).

**Proof:** Let \( j \in \text{Electors}_q \). If the sender of \( j \) is in \( \text{Accept}_p \), then surely \( p \) waits for this message before delivering the join configuration change. Therefore, using Lemma 5.6, \( j \in \text{Electors}_p \). Otherwise, \( j \)'s sender is in \( F_{after}^p \). If any machine in \( \text{Accept}_p \) received \( j \) before acknowledging \( F_{after}^p \), then \( p \) will recover \( j \) (if necessary) and have \( j \in \text{Electors}_p \). Otherwise, all the machines in \( \text{Accept}_p \) acknowledge all of \( F_{after}^p \) before receiving \( j \), and they all agree to discard it from the DAG, in contradiction to \( j \in \text{Electors}_q \). \( \square \)

**Theorem 5.8** Let \( p, q \) be machines. Assume that \( p, q \) deliver their next JOIN configuration change message \( \text{CC}_p, \text{CC}_q \) respectively, s.t. \( p \in \text{Accept}_q \) and \( q \in \text{Accept}_p \). Then:

1. \( \text{CC}_p = \text{CC}_q \).
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 \( \text{Electors}_p = \text{Electors}_q \), which further shows that the Expected array for the CCS is the same in \( p \) and \( q \). \( \square \)

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 \( \text{CC}_p, \text{CC}_q \) respectively, s.t. \( p \in \text{Accept}_q \) and \( q \in \text{Accept}_p \).

1. \( p, q \) deliver the same set of messages following Expected before delivering the CC message.
2. \( \text{CC}_p = \text{CC}_q \)
3. \( \text{Expected}_p = \text{Expected}_q \) 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 asserts 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 \( m_j \in DAC_p \) 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 \( \alpha \) 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 growth 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 straightforward; 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


This article was processed using the \LaTeX\ macro package with LLNCS style.