An Asynchronous Membership Protocol that Tolerates Partitions

Danny Dolev* Dalia Malki† Ray Strong‡

Abstract

This paper presents a membership protocol for maintaining the set of operational and connected machines in agreement. The protocol operates in an asynchronous environment prone to crash failures, omission failures and network partitions. The protocol is suitable for systems with machines that communicate via broadcast (or multicast) messages. It supports continued operation with partitions and provides the mechanism for merging of partitions. The principles of the protocol presented here have been successfully incorporated into the Transis system [3, 2], the Totem system [4], and the Horus system [29].

The membership protocol presented here is integrated in the communication system, such that the notifications of membership changes are delivered to the application among the stream of regular messages. Changes to the membership are coordinated with the delivery of regular messages in the system. This valuable approach was presented in [7, 9] in the context of a primary-partition system, and is extended here for partitionable systems. Its importance is in providing the distributed application builder with a virtually synchronous programming environment. This approach is suitable for many environments today, all of them supporting various forms of multicast communication among machines and among process groups.

Categories and Subject Descriptors: C.2.2 [**Computer Communication Networks**]: Network Protocols—*Protocol Architectures*; C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed Applications*; D.4.5 [**Operating Systems**]: Reliability—*Fault Tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Systems*

General Terms: Distributed Algorithms, Fault Tolerance

Additional Key Words and Phrases: Partitions, Membership, Asynchrony

1 Introduction

This paper presents a membership protocol for maintaining the set of operational and connected machines in agreement. The protocol operates in an asynchronous environment prone to *crash* failures, *omission* failures and network *partitions*. The protocol is suitable for systems with machines that communicate via broadcast (or *multicast*) messages. It supports continued operation

*Institute of computer science, The Hebrew University of Jerusalem, Israel
†Institute of computer science, The Hebrew University of Jerusalem, Israel
‡IBM Almaden Research Center, San Jose, USA

with partitions and provides the mechanism for merging of partitions. The principles of the protocol presented here have been successfully incorporated into the Transis system [3, 2], the Totem system [4], and the Horus system [29].

The membership protocol presented here is integrated in the communication system, such that the notifications of *membership changes* are delivered to the application among the stream of regular messages. Changes to the membership are coordinated with the delivery of regular messages in the system. This approach is suitable for many environments today, all of them supporting various forms of multicast communication among machines and among *process groups* [10, 6, 5, 8, 26, 15, 27, 20, 13, 12, 22, 16, 30, 3].

The problem of maintaining a consistent view of the reachable operational machines is fundamental in the design of distributed systems. In [11], a formal definition of the problem (and solution) is given for synchronous systems. To the best of our knowledge, the first formal definition of the requirements of membership in asynchronous environments is given in [28]. Their paper defines a *primary-partition membership service*, that maintains the *local views* of all the operational machines in agreement. Since it is well known that reaching agreement in asynchronous environments is impossible [14], the membership agreement must circumvent this difficulty somehow. The membership protocol in [28] uses an inaccurate failure detector, based on timeout: When a machine is presumed *faulty*, it is taken out of the view, such that further messages from the faulty machine are *discarded*. In reality, failure detection can be fine-tuned to avoid false-detection almost entirely. This approach is practical, and we adopt it, with an important modification: a presumed failed machine can rejoin the membership, and does not need to *give up*.

In the application requirement in [28], at most one partition may exist, and machines outside the primary partition are either dead or give up. In large and critical systems, this approach is not realistic, and it is essential to enable operation in face of partitions. In the case that partitioned operation is not desired, it is easy to add a layer on top of our membership that disallows all but one partition to operate. Thus, our protocol does not mandate partitions, but provides more flexibility.

We provide a formalization of the requirements of partitionable membership service. When the network partitions, each partition continues operating separately. The machines in each partition are in agreement about membership among themselves, but not with the machines in other partitions. This might sound chaotic, at first. However, we require that:

- Every pair of machines that go through two consecutive membership changes, receive the same set of messages between the two changes. (This is a generalization of the principle called *virtual synchrony*, see [5]).
- Upon re-merging, all the machines in the new membership start with a consistent view of the membership, and agree on the messages that follow it.

In this way, the membership service associates a membership-context with each message. The application can use this to perform consistent operations on received messages, and to merge the histories of joined partitions. For example, [1] describes a replicated mail server that exploits the Transis membership for efficiently implementing a partitionable service.

Intuitively, at the basis of our membership protocol, there are two stages: *(1) suggest a new membership set, (2) wait for agreement from every machine in the set.* This simple protocol is complicated by the following issues:

- All the machines in a new membership set need to terminate the preceding membership in a consistent way. The problem is that if a certain machine, q, is taken out of the previous membership, the adversary might cause the "last" message from q to reach only a subset of the new membership.

  We use the following rule: Let m_q be a message from q. If **any** machine in the new membership set receives m_q before committing to the new membership, **all** of them deliver m_q. Otherwise, they discard it.

- Our design allows the flow of regular messages to continue during transition to a new membership. This design goal is important for handling cascading membership changes, during periods of instability or frequent changes. It also makes the protocol flexible to support messages from external sources (an *open* group communication), because auxiliary sources are not part of the protocol and cannot cease sending messages during its operation.

  In our protocol, the context of regular messages that are sent during membership transitions is determined by their order with respect to the messages used within the membership protocol.

- While forming agreement on a new membership set, there may be additional failures, as well as recoveries or reconnections. One of the questions that motivated our work from the outset was whether "cascading" joinings and failures can be handled in a uniform manner.

  The protocol we present does not eject from the protocol when new failures or joinings occur. Rather, it handles membership changes in a pipeline of agreements, and lets the order among messages determine the order among the changes.

The solution we present supports partitioned operation and rejoining. Joining is done multiway, in a completely symmetrical fashion. In this way, the joining provides a solution to the *startup* problem as well: Each machine starts up as a singleton set on its own. Then, all the machines that start up *merge* into a larger set. Note that our protocol uses broadcast communication, and therefore the symmetry does not incur any additional communication overhead.

The essentials of the membership protocol presented here were implemented within several environments: A full implementation was done within the Transis system [3]. The Totem system [4] is based on a *ring* structure among the connected machines, and its underlying membership protocol is based on the principles of our membership protocol. A partial implementation of the merging algorithm was done within Horus [29]. The porting of the same principles within different environments proves that the protocol is indeed suitable for many multicast environments, and can be implemented cleanly within a separate module.

The contribution of this paper is in giving a formal specification of the membership problem allowing partitions; in providing a symmetrical solution that does not require to "wrap" each change in a single activation, but allows several changes to be processed in pipeline; and in discussing the utilization of continued operation with partitions.

1.1 Motivation for Partitionable Operation

An important novel aspect of the membership service presented in this paper is the tolerance of network partitions. In many similar systems, only one set (the *primary*) of machines is allowed to operate in the system, *e.g.* [28, 23, 24, 11]. Our approach is motivated by the assumption that

the membership service is part of a multicast communication substrate. As such, the membership service must not prevent the upper level applications from continuing operation in face of partitions. Rather, it must provide the upper level with accurate information about the situation in the system, in the same way that point-to-point transport protocols provide the application with information about the status of the connection. This policy allows the application to determine whether to continue operation in one or more of the partitions.

There are several advantages to our approach, that allows partitionable operation:

- There are various ways in which applications can benefit from partitioned operation without a primary partition. The idea is that although there is no primary partition at any given moment, information can gradually diffuse in the system whenever the connection is possible. For example, a system that contains four machines, A, B, C, D, might shift between a ($\{A, B\}$, $\{C, D\}$) configuration and a ($\{A, C\}$, $\{B, D\}$) configuration. In this case, any information that was exchanged between A and B within the first configuration can reach the entire system during the second configuration. Partitionable operation may succeed in diffusing the information gradually, whereas preventing operation in both partitions would prevent any progress in the system.

- Some applications can make do with relaxed consistency guarantees, such that actions may be committed within disconnected partitions. A good example of this type is a network of Automated Teller Machines (ATMs). The actions performed by an ATM, such as the withdrawal of money from a certain account, need to be performed remotely on the bank's main computer, in some accounts database. In today's life, unavailability of ATM service in case it is disconnected from the main computer could be a major impediment. Rather, in case of connection failure, the ATM could be programmed to allow a withdrawal of up to a certain static limit, regardless of the account balance. This scheme is indeed employed in reality.

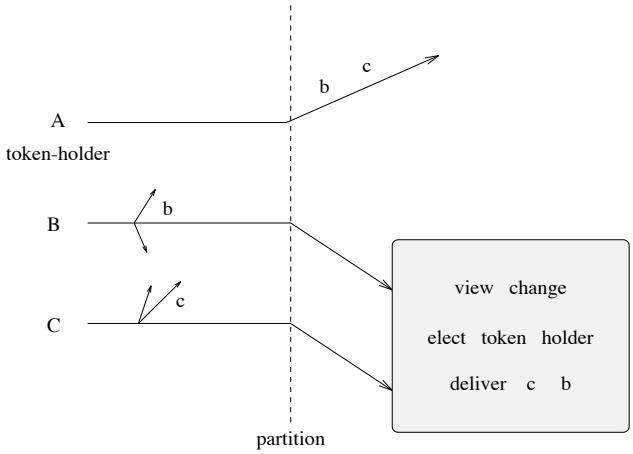

Figure 1: Virtual Partition: A Bad Side Effect

- Good partitionable semantics avoid the inconsistencies that arise when there is no definition of what happens during partitioned operation. As an example of a "bad" side effect typical

in primary-partition systems, consider the scenario in Figure 1. This example shows what can happen during partitions, in a primary partition system with a designated *token holder*. The token holder serves to set an order on messages. In the initial configuration $\{A, B, C\}$, machine A is the token holder, and B and C emit messages b, c respectively. Following that, a virtual partition occurs, and A detaches from B and C. While A has not detected the failure yet, it proceeds to set the order on the messages, and orders b before c. Eventually, A will detect the failure and, since it is isolated in a minority partition, will cease operation. In the other partition, B and C will reconfigure, and will elect a new token-holder (say, B). However, failing any guarantees regarding partitionable operation, B might proceed to order message c before b. The point of this example is to demonstrate that typically, since it is assumed that partition $\{B, C\}$ will prevail, no concern is given to what happens in partition $\{A\}$. However, this may lead to visible inconsistencies. Therefore, it is important to define the semantics of communication operations during partitions, even if only the primary partition is allowed to continue.

- There are situations in which the system cannot maintain a primary partition, such as when a majority of the machines fail, or when the system splits into multiple small partitions. This difficulty can be proved rigorously: the impossibility of maintaining a primary partition in agreement in an asynchronous system with faults stems from the impossibility of consensus in asynchronous systems [14]. In some systems, the situations in which there is no primary partition will be common, *e.g.* in wireless networks. Therefore, for these systems, the membership service should provide the option to continue operation within non-primary partitions.

- At startup time, a distributed system may have several disconnected partitions. In the startup procedure, each partition might form its own membership set. The only way to avoid multiple partitions forming at startup is to use an external source of agreement, such as a central name-server (as in *e.g.* [29, 28]), or use a predetermined configuration. However, allowing for multiple partitions to exist, the most natural way to start up the system is to let each machine form a *singleton set*, and then *merge* the sets into larger sets whenever the connection is possible. As all the potential sets in the merging are active, the merging is symmetrical: There is no active or primary side, and no joining side.

2 The System Model

The system comprises of a finite universe of *machines* that communicate via asynchronous multicast messages. A multicast message leaves its source machine at once to all the target machines in the system but may arrive at different times to them. Messages are uniquely identified through an ordered pair < sender, counter > [1].

Several kinds of faults can occur in the system: machines may incur a crash fault, and may restart. The communication network(s) might partition and re-merge. When the communication network partitions, massages might be lost between the partitions. Faults cannot alter messages' contents. When there is no fault, messages arrive to their destinations reliably. However, messages might be delayed arbitrarily long.

[1] In order to guarantee this uniqueness, we assume that the machines write on stable storage an incarnation number, that is part of the message identification. The incarnation field is inceremented upon startup.

It is sometimes convenient to model the system as a collection of finite-state machines. Each machine consists of internal-state data and a transition function. The behavior of the system is recorded via a history of state-transition events occurring at each machine. We denote the history of the system as a linear list of events $H = e_1, e_2, \ldots$ (H is written by a "by-stander" observing the events as they occur; when two events occur simultaneously, the bystander determines arbitrarily which one is written first). The history H induces a precedence order $\xrightarrow{H}$ on the events of H.

The first event by any machine p in the history H is a special event, denoted $start_p$. A history H contains an infinite number of events by machine p, or a finite number if p crashes. For the simplicity of the exposition, we add an artificial event that terminates the history of a crashed machine, denoted $crash_p$. This event is not taken by p itself, nor is it visible to other machines within the system.

2.1 The Communication Model

Each machine in the system contains a *transport-layer* that handles the group communication operations. We model the transport layer in each machine as consisting of two modules: a *causal communication module* (or simply a *communication module*) and a *membership module* on top of it. Figure 2 depicts the transport layer structure. The membership protocol implements the membership module.

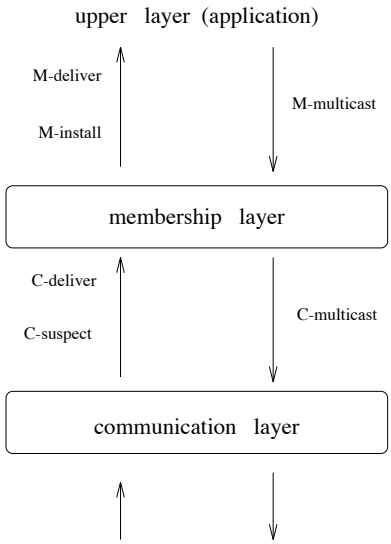

Figure 2: The System Model Structure

The user application operates on top of the transport layer. The transport layer accepts messages for multicast from the layer above it, and delivers messages up for receiving. In addition to application (regular) messages, there may be messages that are generated internally by the transport-layer modules, *e.g.* by the membership protocol. To be precise, we distinguish between events that are concerned with different modules. At a machine p , we denote the following events:

- $\mathcal{M}$–multicast$_p(m)$: The membership module is requested to multicast message m. The mes-

sage m includes the sender's identification, p.

- $\mathcal{C}$–multicast$_p(m, \text{targets})$: The communication module is requested to multicast a message m to the list of target machines in *targets*. In the context of this paper, all the messages are multicast to the same set of machines, the universe (S). Below, the *targets* field is omitted when it is S.

- $\mathcal{M}$–deliver$_p(m, q)$: The memb module delivers a message m sent by q. When the sender is unimportant to the context, we write $\mathcal{M}$–deliver$_p(m, *)$, or in short $\mathcal{M}$–deliver$_p(m)$.

- $\mathcal{C}$–deliver$_p(m, q)$: The communication module delivers a message m sent by q. Likewise, when the sender is unimportant, we write $\mathcal{C}$–deliver$_p(m, *)$, or in short $\mathcal{C}$–deliver$_p(m)$.

The communication module makes its best effort to guarantee the reliable delivery of multicast messages to all the targets. It notifies the layer above when problems occur, *e.g.* when a message cannot be transferred to some targets. The communication module does not attempt to handle such failures or recover in any way, but simply passes up a failure-notification:

- $\mathcal{C}$–suspect$_p(q)$: the communication module notifies that q is "suspected" to be detached from p. Note that this is a different event from the *install* event (below) that establishes a new membership without q; the *install* event occurs only later, after the membership protocol reaches agreement on it.

The membership module maintains agreement about the system configuration. It reports to the user application about changes in the membership via special events:

- $\mathcal{M}$–install$_p(\mu)$: the membership module at p delivers a membership installation message μ. μ contains the new membership-set of machines, and a unique *context* field. We denote by $\overline{\mu}$ the machine-set alone. A $\mathcal{M}$–install$_p(\mu)$ event is unique in the history of a machine p (whereas the set $\overline{\mu}$ of machines, without the context field, may be installed multiple times).

- $\mathcal{M}$–scatter$_p(\mu)$: the membership module at p reports of a problem with the membership μ. This event can occur between successful install events, during periods of instability. It indicates to the user that an intermediate attempt to install a new membership, μ, has failed.

We define after Lamport [21] the *causal* order of messages, as induced by the order of events in a history H, as the reflexive, transitive closure of:

(1) $m \xrightarrow{cause} m'$ if $\mathcal{C}$–deliver$_q(m) \xrightarrow{H} \mathcal{C}$–multicast$_q(m', *)$

(2) $m \xrightarrow{cause} m'$ if $\mathcal{C}$–multicast$_q(m, *) \xrightarrow{H} \mathcal{C}$–multicast$_q(m', *)$

Note that any two multicast events occurring at the same machine are ordered by H, and so $\xrightarrow{cause}$ preserves the order of transmission by each machine. In addition, $\xrightarrow{cause}$ preserves any potential causality.

Below we state the properties that are assumed about the communication module, by constraining the histories of events in the system.

The first property requires that messages are delivered by the machines in an order that preserves potential causality:

Requisite C.1 (Causal prefixes) *Let $e \equiv \mathcal{C}$–deliver$_p(m)$ be an event of machine p in H. Let m' be a message s.t. $m' \xrightarrow{cause} m$. Then $e' \equiv \mathcal{C}$–deliver$_p(m')$ is an event of H, and $e' \xrightarrow{H} e$. Moreover, there are no nontrivial cycles in $\xrightarrow{cause}$.*

Note that requisite C.1 implies that a newly-joined machine must be able to recover the complete history of messages in the system. In practice, this might pose a heavy storage burden. Indeed, the amount of history-messages that are actually required for the correctness of the membership protocol is bounded. Therefore, in reality, the causal histories are trimmed. However, for simplicity of the exposition, we ignore this issue and assume complete causal-prefixes. Also, we assume that the authentication and error detection properties of the communication layer are strong enough to prevent apparent causality paradoxes where, for example, message m is $\mathcal{C}$–delivered before m' is $\mathcal{C}$–multicastat p but m' is $\mathcal{C}$–delivered before m is $\mathcal{C}$–multicastat q.

The next two requirements concern the detection of faults in the system. Fault detection in an asynchronous system can never be *precise*. Typically, one uses *timeout* to identify when the communication with other machines has possibly been disrupted. The detection is done locally and independently at each machine. In this way, machines may be suspected to be detached at one time, and later removed from suspicion. There is no way of verifying a suspicion beyond all doubt. However, in practice, we can usually assure that if a machine has crashed or disconnected, then the other machines will detect it within a finite amount of time. Also, in practice, we can usually guarantee that if the communication system delivers all our messages promptly, then there will be no erroneous suspicions (otherwise, a fault detector that simply suspects everybody all the time will be trivially correct, but useless).

Requisite C.2 (Liveness) *For each event $e \equiv \mathcal{C}$–multicast$_p(m,$*targets*$)$ in the history H, there is a point in H by which for each $q \in$* targets*, there is an event e' at p, $e \xrightarrow{H} e'$, such that:*

- *$e' \equiv \mathcal{C}$–deliver$_p(m_q, q), m \xrightarrow{cause} m_q$, meaning that p receives an implicit acknowledgment from q, or*
- *$e' \equiv \mathcal{C}$–suspect$_p(q)$,* i.e. *p is notified by the communication module of a possible disconnection from q, or*
- *$e' \equiv crash_p$* i.e. *p crashes.*

Requisite C.3 (Non Triviality) *Let* T *be a set of machines, and $\{<p_1, m_1>, <p_2, m_2>, \ldots\}$, $p_i \in T$, be an arbitrary finite set of pairs. Then there exists an execution history H, in which for every pair $<p, m>$ in the set, p successfully multicasts message m to* T*;* i.e. *there exists $e \equiv \mathcal{C}$–multicast$_p(m, T) \in H$, there is no event crash$_p$ in H, and for every q in* T*, there is no event $\mathcal{C}$–suspect$_p(q)$ in H following e.*

Henceforth, we assume that the causal communication module in our environment satisfies Requisites C.1, C.2, C.3.

2.2 Causal delivery and Transis

The causal communication properties C.1, C.2, C.3, are currently supported in many systems, *e.g.* Transis [3], Isis [8], Psync [27], Horus [29], and the Trans protocol [22]. The common factor to all of these systems is the relatively low overhead costs. In this subsection we give some details of the causal multicast in Transis.

Transis contains the communication module responsible for the reliable and causal delivery of multicast messages in the system [3]. In Transis, each newly emitted message contains ACKs to previous messages. The ACKs form the $\stackrel{cause}{\longrightarrow}$ relation directly, such that if m' contains an ACK to m, then $m \stackrel{cause}{\longrightarrow} m'$. The Transis communication sub-system receives the messages off the network. If a message arrives at a machine and some of its causal predecessors are missing, Transis transparently handles message recovery and re-ordering. It performs recovery and message handling, and at some later time, it passes the messages to the upper level (producing the $\mathcal{C}$–deliver events described in the previous section). The multicast communication module of Transis is the motivation for our model definition, and provides the basis for devising the membership protocol.

3 The Membership Service

The membership module maintains agreement about the system configuration, and reports about changes in the membership via $\mathcal{M}$–install events. We make several requirements of the generation of install events that reflect the membership problem in our environment. Our definition allows partitions to occur, and defines the behavior despite them.

The first requirement refers to the order of membership changes at different machines:

Requisite M.1 (Order) *Let $\mathcal{M}$–install$_p(\mu)$, $\mathcal{M}$–install$_p(\mu')$, be two consecutive install events at p, and let $\mathcal{M}$–install$_q(\mu)$, $\mathcal{M}$–install$_q(\mu')$ be events in H. Then*

$$\mathcal{M}\text{–install}_p(\mu) \stackrel{H}{\longrightarrow} \mathcal{M}\text{–install}_p(\mu') \Rightarrow \mathcal{M}\text{–install}_q(\mu) \stackrel{H}{\longrightarrow} \mathcal{M}\text{–install}_q(\mu') \ .$$

In words, Requisite M.1 means that machines that go through the same membership changes, go through them in the **same** order.

The removal of detached or faulty machines from the membership is essential for the liveness (termination) of the membership protocol. Recall that the communication layer notifies of (possible) communication failures via $\mathcal{C}$–suspect events (Requisite C.2). These events are internal, and are not reported to the upper layer. There is no direct requirement of the membership service to use the $\mathcal{C}$–suspect events internally. However, Requisite M.2 below guarantees that if a machine q appears quiet for a certain duration, then it is removed from the membership within a finite time. In turn, this implies that the membership protocol that implements the membership maintenance is non-blocking.

Requisite M.2 (Termination) *Let p send a message in event $e \equiv \mathcal{M}$–multicast$_p(m)$ in the history H, such that the last install event in p preceding e is $\mathcal{M}$–install$_p(\mu)$. Then there is a point in H by which $\forall q \in \overline{\mu}$, there is an event $e' \in H$ following e, such that:*

- $e' \equiv \mathcal{C}$–*deliver*$_p(m_q, q), m \stackrel{cause}{\longrightarrow} m_q$, *meaning that* p *receives an implicit acknowledgment from* q, *or*
- $e' \equiv \mathcal{M}$–*install*$_p(\mu')$, *s.t.* $q \notin \overline{\mu'}$, i.e. p *removes* q *from the agreed-upon membership, or*
- $e' \equiv crash_p$, i.e. p *crashes.*

A trivial membership protocol that lets each member form a singleton set, solves the membership problem defined so far. The following non-triviality requirement rules out such solutions:

Requisite M.3 (Non-Triviality) *Let* T *be a set of machines. There exists a history* H, *and an event* $\mathcal{M}$–*install*$_p(\mu)$ *in* H, *such that the installed membership-set in* μ *is* T (i.e. $\overline{\mu}$, *the membership-set of* μ, *is* T).

Recall that the membership module also filters regular messages to and from the application. The membership changes ($\mathcal{M}$–install events) are delivered to the upper level application among the stream of regular messages. The membership protocol interferes with the delivery of regular and membership messages to the next module above. This reflects our belief that the membership maintenance should be met at a low level, to provide context for messages. In distributed applications, the programs running on different machines can act upon messages according to the context. Thus, it is desired that the machines see the same messages between every pair of membership changes. This valuable principle is defined in [9], and is called *virtual synchrony*. The next requisite expresses the demands on the relative order of message delivery and installations of membership changes:

Requisite M.4 (Virtual-Synchrony)

(1) *Let* $\mathcal{M}$–*deliver*$_p(m)$, $\mathcal{M}$–*deliver*$_p(m')$ *be two events in the history* H. *Then:*

$$m \stackrel{cause}{\longrightarrow} m' \Longrightarrow \mathcal{M}\text{–}deliver_p(m) \stackrel{H}{\longrightarrow} \mathcal{M}\text{–}deliver_p(m') \ .$$

(2) *Let* $\mathcal{M}$–*install*$_p(\mu)$, $\mathcal{M}$–*install*$_p(\mu')$, *be consecutive install events at* p, *and let* $\mathcal{M}$–*install*$_q(\mu)$, $\mathcal{M}$–*install*$_q(\mu')$ *be events in* H. *Then*

$$\begin{aligned} \forall m: \ \ & \mathcal{M}\text{–}install_p(\mu) \stackrel{H}{\longrightarrow} \mathcal{M}\text{–}deliver_p(m) \stackrel{H}{\longrightarrow} \mathcal{M}\text{–}install_p(\mu') \Rightarrow \\ & \mathcal{M}\text{–}install_q(\mu) \stackrel{H}{\longrightarrow} \mathcal{M}\text{–}deliver_q(m) \stackrel{H}{\longrightarrow} \mathcal{M}\text{–}install_q(\mu') \ . \end{aligned}$$

(3) *Let* $\mathcal{M}$–*install*$_p(\mu)$ *be the last install event at* p *preceding* $\mathcal{M}$–*deliver*$_p(m, s)$. *Then* $s \in \overline{\mu}$.

In primary-partition membership services, when a machine p installs a certain membership, it must become the unique valid membership of the system. However, what does it mean for a machine p to install a membership μ in a partitionable system? After all, during transient periods of instability, machine p cannot guarantee that all the other machines in μ will install

the membership μ, nor can it guarantee that the members that do not install it *give up*, as in a primary-partition membership service. The membership service of [4] discusses the installation of a *transient membership* in these cases. It is beyond the scope of this paper to discuss the importance of this information for application builders, and we refer the interested reader to [4, 25]. We do require that if p installs μ, all the other members in μ will at least be able to report of a possible membership change at that point (or crash).

Requisite M.5 *Let $\mathcal{M}$–install$_p(\mu)$ be an event of H. Let q be a machine in $\overline{\mu}$. Then there exists one of the following events:*

- *crash$_q \in H$, or*
- *$\mathcal{M}$–scatter$_q(\mu) \in H$,* i.e. *q delivers an event notifying of a problem with the installation of μ, or*
- *$\mathcal{M}$–install$_q(\mu) \in H$,* i.e. *q also installs μ.*

3.1 Related Work

The maintenance of the membership of machines is a basic building block of many fault-tolerant applications. It is interesting to compare (some of) the works done on the membership problem in view of the requisites that are formulated in the previous section.

Most membership algorithms fall into the category of primary-partition memberships: they demand that there is a unique sequence of view changes of the membership throughout the system. Our membership service allows partitioned operation, and most of the differences are derived from this:

- Requisite M.1: All of the membership services we are aware of require the total ordering of *installed* membership changes, expressed in Requisite M.1.
- Requisite M.2: All the membership mechanisms we are aware of guarantee that a faulty machine is taken out of the membership within a finite time, similarly to Requisite M.2.
- Requisite M.3: How close does the membership resemble "reality"? In primary partition membership protocols, the desire of the membership to resemble reality is expressed in the desire to include all the *correct* machines in the membership. For example, in [23], a correct machine that performs infinitely many *attempts* to join the membership must succeed with probability one. The synchronous membership protocol of [11] places a time-bound on the amount of time it takes a correct machine to join the membership, provided that it remains correct throughout this period. Ricciardi et al. [28] simply demand that machines are not taken **out** of the membership capriciously. Since they make no constraints on the fault detection in the system, their requirement does not guarantee that any two machines will ever form a joined membership.

  In partitioned systems, the notion of correctness of machines is meaningless, because two "correct" machines may nevertheless be disconnected. All we demand in Requisite M.3 is that there exist circumstances (extreme as they may be, for all we know) that allow a certain set of machines to form a membership.

- Requisite M.4: The virtual synchrony principle of Requisite M.4 has been adopted by several membership systems [2, 28, 4] and is also present in [23] (which totally orders all the messages, including membership changes). However, it is not supported by all: the membership protocol in [24] preserves causal-order between membership changes and regular messages, but does not guarantee virtual synchrony. There are several protocols that provide a membership service that is not related to multicast message delivery, such as [17, 11].

- Requisite M.5: In a primary partition membership, *e.g.* [28], whenever a membership is installed, it will become the (only) view of the system at all the operational machines. Another example is the membership algorithm of [23], which is based on a single total order of messages throughout the system. Their membership changes are induced by the unique sequence of messages, and therefore are also unique.

  Since we allow partitions to occur, we must confine such a commitment to within the set of *connected machines*. In Requisite M.5, we capture the notion that two machines p, q will go through the same sequence of membership changes until one of them crashes or until they detach from each other.

  It is possible to relax Requisite M.5 altogether: in the Weak Membership of [17], there is no guarantee that all the members see a certain membership installation. Their protocol simply assures that if the communication is timely and there are no faults, then the membership will be in consensus. In periods of instability, the membership view might diverge at different machines. For purposes such as object replication, this membership is too weak (indeed, [17] also provides the Strong and the Hybrid memberships for consistent replication).

Another issue is the possibility of a machine to return to the membership after being disconnected. In [28], it is assumed that a machine that is taken out of the membership never returns to it. This implies that if a machine is mistakenly taken out of the membership, then it gives up when it finds out that it was taken out of the membership. When partitions are supported, this assumption can no longer hold: a machine may be removed from some membership but remain active in a different partition. Upon reconnection, none of the partitions will give up. Thus, machines that are removed from the membership may rejoin it later. Care must be taken for the continuity of messages to and from rejoined machines: We assume that each machine writes on stable storage an *incarnation* number, that is used as part of messages' identification. When a machine crashes and recovers, the incarnation number is incremented and thus, the message-id's it generates are unique. For simplicity, we assume that when machines join, they retransmit the complete causal history of messages, such that at both partitions there are no causal "holes". (In practice, methods for garbage-collecting and trimming of saved histories are obviously applied).

There are other membership services that allow partitioned operation. The membership service of the Amoeba system [20, 19] lets the user determine the minimal size with which the system can continue operating. If the user determines a majority threshold, the result is a primary-partition membership service. On lower thresholds, the system may partition. The user has the control in trading resiliency with consistency. The protocol presented in [18] does not provide any solution for merging operational partitions upon reconnection.

Related solutions to the membership problem that support partitions appear in [2, 4]. The membership protocol presented here differs in being active throughout any number of membership changes. Rather than ejecting from activation after each membership change, it cascades through

them and lets the causal order of messages determine the order of changes. The protocol in [2] supports a restricted version of Requisite M.4.

4 The Membership Protocol

This section presents a protocol that implements the membership service.

4.1 Notation

The membership protocol uses special messages, called *membership messages*, of the form <P; F; *context*>, containing a membership set P, a faults set F, and a unique context-field. This membership message stands for suggesting the membership

$$\overline{\text{<P; F; } context\text{>}} \equiv \text{P} \setminus \text{F} .$$

Membership messages are broadcast to the entire universe of machines, S. Although we do not mention this explicitly in the protocol, we assume that periodically, the machines broadcast "probe" messages (that can be NULL), in order to learn about other machines that have joined the configuration. In this way, the machines may receive "foreign" messages.

Let $\mu =$ <P; F; c> be a membership message. We use the following notations:

$\overline{\mu}$ – Denotes the *membership set* indicated by μ, P \ F .

EQUIV$_p(\mu)$ – Denotes a message from machine p whose P, F, and c are identical to that of message μ. This notation will be useful below, when we discuss scenarios in which different machines multicast identical messages (and thus "agree" on the membership contents). We do not write a simple equal sign in this case, to stress the difference between messages from different senders, even if they have identical contents.

UNIFORM(μ) – The set of all EQUIV$_r(\mu)$ messages, such that $r \in \overline{\mu}$.

CCONE(μ) – The set of messages that causally precede *any* message in UNIFORM(μ). We call this the *causal cone* of UNIFORM(μ).

4.2 Data Structures

Each machine p maintains the following data structures:

Memb – The currently installed membership-set (of machines).

AM,AF,context – AM is a membership set, containing the accumulated set of machines that are expected to form the next agreement on membership. AF is a set containing the accumulated set of suspected faults. *context* is an integer variable that is incremented every time a new membership is installed. AM, AF are working variables, that accumulate machines until a new membership is installed.

$\mathcal{MQ}^i$ – For each machine $i \in AM$, $\mathcal{MQ}^i$ is a queue containing all the pending membership messages received from i. The queues $\bigcup_{i \in AM} \mathcal{MQ}^i$ are used for maintaining all the received membership messages, until a decision to install a certain membership is made. We denote the k'th message in the queue $\mathcal{MQ}^i$ with $\mathcal{MQ}^i[k]$.

delay$\mathcal{Q}$, *pending*$\mathcal{Q}$ – A queue of delayed regular messages, and a queue of messages pending delivery. These queues are used as follows: The protocol inserts each $\mathcal{C}$–delivered message either into *delay*$\mathcal{Q}$ or into *pending*$\mathcal{Q}$. It delays messages in *delay*$\mathcal{Q}$ and delivers messages from *pending*$\mathcal{Q}$ in their causal order. Membership messages pending in *pending*$\mathcal{Q}$ are initially marked *nondeliverable* and are blocked from delivery. Consequently, messages that causally follow a membership message are also blocked from delivery. When the protocol decides to install or reject a membership message, it marks it *install*, *reject* or *scatter*. Subsequently, it delivers all the messages that causally follow the membership message.

The message queues used by the protocol and a schematic picture of the flow of messages among them are depicted in Figure 3.

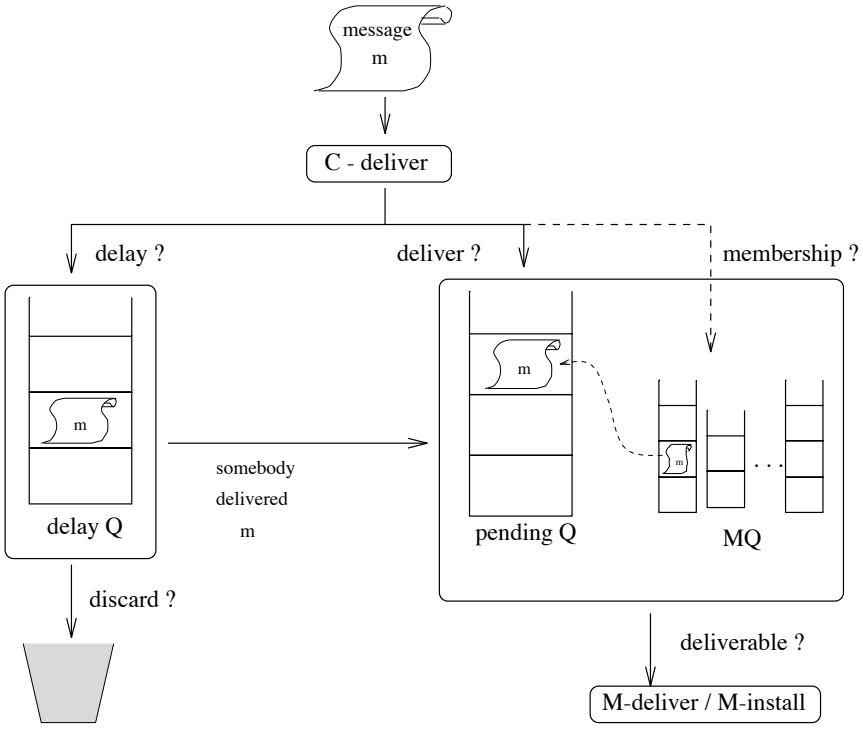

Figure 3: The main data structures of the protocol

4.3 The Protocol

The protocol is described in English below. A formal description in pseudo-code is provided in Figures 4–8.

The main loop

Each machine performs an infinite loop, handling communication layer events. Figure 4 gives the main program performed by machine p in pseudo-code. Initially, machine p starts with a membership set containing itself only. The main loop at p handles the following events:

- When the communication layer reports about a problem (via $\mathcal{C}$–suspect), p adds the suspected machine to AF, and multicasts a membership message with the new AF set.
- When the communication layer $\mathcal{C}$–delivers a membership message from a suspected machine ($\in AF$), then p places it in $delay\mathcal{Q}$, and sends a membership message with the current AM, AF sets.
- When the communication layer $\mathcal{C}$–delivers a membership message from a non-suspected machine ($\notin AF$), then p invokes HANDLE-MEMBERSHIP().
- When the communication layer $\mathcal{C}$–delivers a regular message from a suspected machine (in AF) or from a joining machine (in $AM \setminus Memb$), p delays it in $delay\mathcal{Q}$.
- When the communication layer $\mathcal{C}$–delivers a regular message from a machine in the current membership, p places it in $pending\mathcal{Q}$, and calls DELIVER().
- When the communication layer $\mathcal{C}$–delivers a "foreign" message, *i.e.* a message from a machine outside $AM \cup AF$, p adds the new machine to AM, and sends a new membership message.

Procedure HANDLE-MEMBERSHIP()

The procedure handles a membership message received by p as follows (pseudo-code is given in Figure 5):

It inserts the membership message into $pending\mathcal{Q}$, and marks it *nondeliverable*. If the membership message is from machine q, a pointer to it is kept in $\mathcal{MQ}^q$. Thus, $\mathcal{MQ}^*$ is a vector of queues (one per machine, including p itself), that keeps references of all the pending membership messages.

If the membership message does not appear in $\mathcal{MQ}^p$, p invokes MERGE-MEMBERSHIP() to merge the information in this message. There is no need to check whether it can be installed: membership messages that do not appear in $\mathcal{MQ}^p$ will not be installed by p, because p did not "agree" to them (yet).

If the received message appears in $\mathcal{MQ}^p$, p performs a loop, deciding about membership messages one at a time, according to their order in $\mathcal{MQ}^p$. The loop searches for the first message in $\mathcal{MQ}^p$ that has **full agreement:**

Definition 4.1 We say that a membership message μ has *full agreement* when $\mathcal{MQ}^*$ contains messages $\text{EQUIV}_q(\mu)$ from every $q \in \overline{\mu}$.

If $\mathcal{MQ}^p[1]$ has full agreement, p updates AM_p, updates AF_p, and increments $context_p$. The machines in the F set of $\mathcal{MQ}^p[1]$ are removed from AM_p, and AF_p is assigned the union of all the faults-sets of pending membership messages. This means that a suspected machine s remains in AF as long as there is any pending membership message that contains s in its faults set. After all

the pending membership messages that contain s in the faults set are handled, s may rejoin the membership.

In order make the installation of $\mathcal{MQ}^p[1]$ deliverable to the user, p invokes UNBLOCK-MEMBERSHIP($\mathcal{MQ}^p[1]$, *install*) . The reason that the $\mathcal{M}$–install event does not take place immediately is that UNBLOCK-MEMBERSHIP needs to order this event with relation to all the regular messages in $delay\mathcal{Q}$ and $pending\mathcal{Q}$, in order to preserve the virtual synchrony rule (Requisite M.4).

If the minimal k such that $\mathcal{MQ}^p[k]$ has full agreement is some $k > 1$, then p checks for **hidden agreement** on $\mathcal{MQ}^p[1]$:

Definition 4.2 We say that $\mathcal{MQ}^p[1]$ has *hidden agreement* if CCONE($\mathcal{MQ}^p[k]$) contains messages EQUIV$_r$($\mathcal{MQ}^p[1]$) from every machine r in $\overline{\mathcal{MQ}^p[1]} \cap \overline{\mathcal{MQ}^p[k]}$.

In the case that $\mathcal{MQ}^p[k]$ has full agreement, if $\mathcal{MQ}^p[1]$ has hidden agreement, p delivers it as *scattered* membership, by invoking UNBLOCK-MEMBERSHIP($\mathcal{MQ}^p[1]$, *scatter*) . Otherwise, p *rejects* $\mathcal{MQ}^p[1]$.

The reason for "aborting" a membership message (via *reject* or *scatter*) is that full agreement on the first message $\mathcal{MQ}^p[1]$ might be impossible, if machines in $\overline{\mathcal{MQ}^p[1]}$ have failed or disconnected from p. However, if a certain machine is nonresponsive, it is eventually $\mathcal{C}$–suspected by p, or by another machine in $\overline{\mathcal{MQ}^p[1]}$. The suspected machine is added to AF, and is included in a following membership message from p, which is queued in $\mathcal{MQ}^p$.

Therefore, in procedure HANDLE-MEMBERSHIP, p continues to examine the membership messages that follow $\mathcal{MQ}^p[1]$ in the queue. The membership messages in $\mathcal{MQ}^p$ do not necessarily cancel each other: p handles multiple pending membership changes. If a certain membership message $\mathcal{MQ}^p[k]$, $k > 1$, already has full agreement, then $\mathcal{MQ}^p[k]$ will be installed by p, and $\mathcal{MQ}^p[1]$ will not be installed.

In order to guarantee Requisite M.5, p must know whether "hidden" agreement on $\mathcal{MQ}^p[1]$ is possible. Obviously, p has already noted that the machines outside $\overline{\mathcal{MQ}^p[1]} \cap \overline{\mathcal{MQ}^p[k]}$ have detached, and p may be lacking agreement from these machines. Thus, hidden agreement is defined within the set of machines that are currently connected to p.

Procedure MERGE-MEMBERSHIP()

If the incoming membership message contains p in its F set, p reverses the suspicion, and adds the sender to AF_p. If the membership message is received from a machine in AM_p, but contains an increased *context*, then p regards this as one-way detachment as well, and again reverses the suspicion. In all other cases, p merges its AM_p set with the message's P set, its AF_p set with the message's F, and takes the maximum of the *contexts*.

After merging, p sends a new membership message.

Procedure UNBLOCK-MEMBERSHIP()

Recall that upon arrival, membership messages are marked *nondeliverable*. In this way, a membership message in $pending\mathcal{Q}$ serves as a place-holder, that prevents messages that causally follow it from being delivered.

Procedure UNBLOCK-MEMBERSHIP() handles the unblocking of membership messages. When it is invoked with an *install* decision, it performs the following operations:

Move to *pending*$\mathcal{Q}$**:** Move all the delayed messages in CCONE(μ), from members in $Memb_p$, to *pending*$\mathcal{Q}$. This is required to support virtual synchrony: if *any* machine in $\overline{\mu}$ delivers a message before the install event, then *all* the machines will deliver it.

Move all the messages outside CCONE(μ), from machines in $\overline{\mu}$, to *pending*$\mathcal{Q}$. In case of cascading membership changes, these "foreign" messages were held in *delay*$\mathcal{Q}$. They become "relevant" once their sender becomes a member of the current membership. This guarantees Requisite M.4(3).

Discard: Discard all the messages outside CCONE(μ) from faulty machines (*i.e.* machines in the F set of μ). These messages will be discarded by all the machines in $\overline{\mu}$. This principle has utmost importance: it enables the protocol to evade the impossibility of agreement. Unless some machine within the membership has delivered a message from a suspected machine **before** the suspicion, the message is discarded by everybody. The idea is that no adversary could schedule such a message to be delivered at "the last minute". This guarantees Requisite M.4(2).

Discard all the messages in CCONE(μ), from machines outside $Memb_p$. These are messages that were delivered in other partitions, before they are joined.

For any value of *decision*, the procedure synthesizes a membership message with identical contents to μ that causally follows all the messages in CCONE(μ). It replaces all the membership messages of UNIFORM(μ) with the synthesized message. This guarantees that the $\mathcal{M}$–install$_p(\mu)$ or $\mathcal{M}$–scatter$_p(\mu)$ events will follow the delivery of all the messages in CCONE(μ), in order to preserve the virtual synchrony.

Procedure DELIVER()

The procedure DELIVER() handles the delivery of regular messages and membership messages (see Figure 8 for pseudo-code). DELIVER() takes messages from *pending*$\mathcal{Q}$, and delivers them in their causal order, thus guaranteeing Requisite M.4(1).

The procedure handles membership messages prior to regular messages[2]. It handles membership messages in the following way:

- Delay messages that are marked *nondeliverable*.
- Install messages that are marked *install*.
- Deliver messages that are marked *scatter*.
- Discard messages that are marked *reject*.

[2]Thus, membership changes are installed before any causally-concurrent messages.

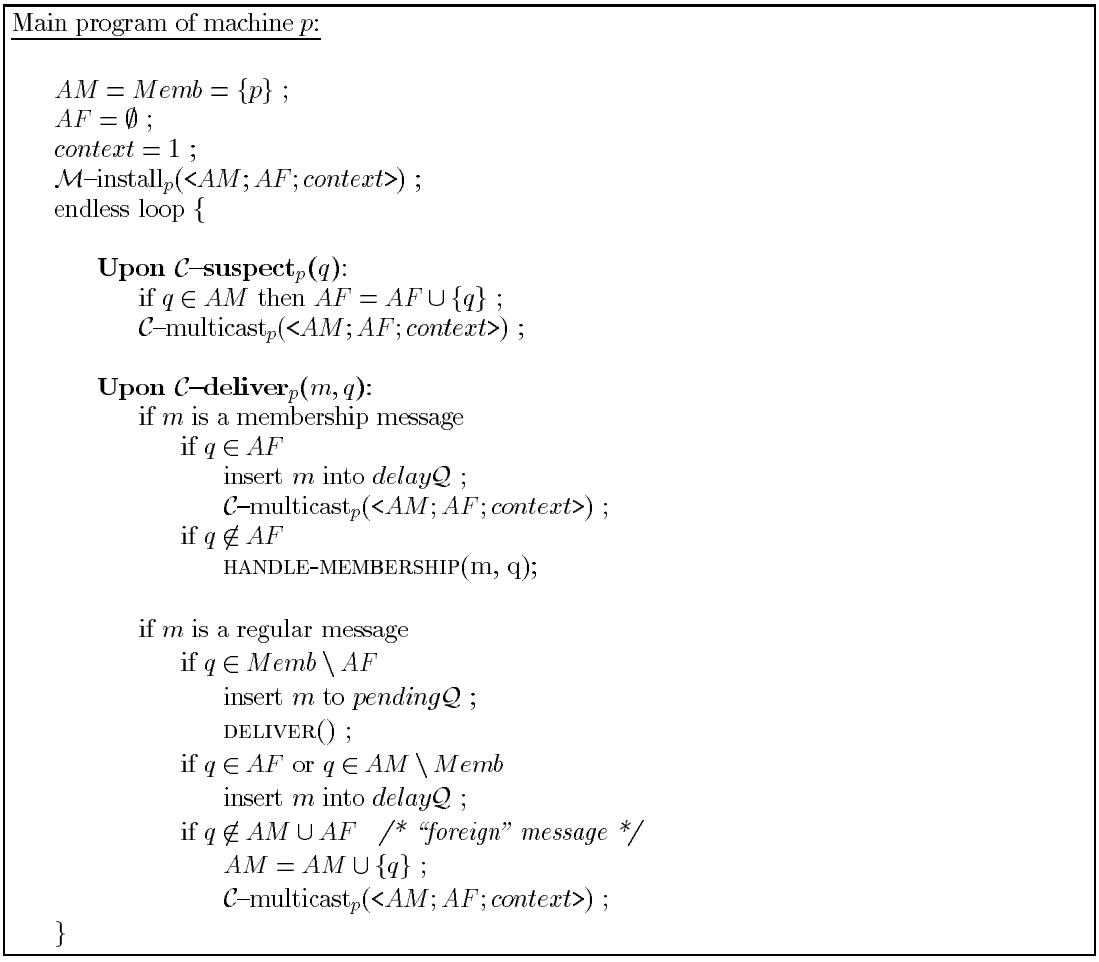

Figure 4: Main loop: Handle communication layer events

```
procedure HANDLE-MEMBERSHIP(<P; F; c>, q):

    insert <P; F; c> into pending𝒬, and mark it nondeliverable ;
    insert a pointer to <P; F; c> into ℳ𝒬^q ;
    if EQUIV_p(<P; F; c>) does not exist in ℳ𝒬^p
        MERGE-MEMBERSHIP(<P; F; c>, q) ;

    if EQUIV_p(<P; F; c>) exists in ℳ𝒬^p then loop until exit {
        Let k be the minimal index such that ∀q ∈ overline(ℳ𝒬^p[k]), ℳ𝒬^q contains EQUIV_q(ℳ𝒬^p[k]) ;
        if k == ∞ then exit loop ;
        if k == 1 then   /* full agreement on ℳ𝒬^p[1] */
            AM = AM \ F ;
            AF = ⋃F^i,  where F^i is the faults–set in ℳ𝒬^p[i], i = 2 .. size-of(ℳ𝒬^p) ;
            increment context ;
            UNBLOCK-MEMBERSHIP(ℳ𝒬^p[1], install) ;
        if k > 1   /* check hidden agreement on ℳ𝒬^p[1] */
            if hidden agreement on ℳ𝒬^p[1] [a]
                UNBLOCK-MEMBERSHIP(ℳ𝒬^p[1], scatter) ;
            else   /* reject ℳ𝒬^p[1] */
                UNBLOCK-MEMBERSHIP(ℳ𝒬^p[1], reject) ;

        ∀r ∈ AM pop messages from ℳ𝒬^r up to and including EQUIV_r(ℳ𝒬^p[1]) (if it exists) ;
    }

[a] Recall that hidden agreement is defined as:
    ∀r ∈ (overline(ℳ𝒬^p[1]) ∩ overline(ℳ𝒬^p[k])) : ∃EQUIV_r(ℳ𝒬^p[1]) ∈ CCONE(ℳ𝒬^p[k]) .
```

Figure 5: Handle incoming membership messages

```
procedure MERGE-MEMBERSHIP(<P; F; c>, q):
    if p ∈ F   /* reverse the suspicion */
        AF = AF ∪ {q} ;
    if q ∈ AM and c > context   /* q has detached */
        AF = AF ∪ {q} ;
    if p ∉ F and (q ∉ AM or c ≤ context)   /* all other cases */
        AM = AM ∪ P ;  AF = AF ∪ F ;  context = max(context, c) ;

    if AM, AF or context changed above then 𝒞–multicast_p(<AM; AF; context>) ;
```

Figure 6: Merge incoming membership messages

```
procedure UNBLOCK-MEMBERSHIP(<P; F; c>, decision):

    if decision is install
        for every message m from q in delayQ  {
            if m ∈ CCONE(<P; F; c>)
                if q ∈ Memb move m from delayQ  to pendingQ ;
                else discard m ;   /*  discard messages from other partitions before the join */
            if m ∉ CCONE(<P; F; c>) and q ∈ F
                discard m ;   /*  discard belated messages from F  */
            if m ∉ CCONE(<P; F; c>) and q ∈ P \ AF
                move m to pendingQ ;   /*  m is relevant to the new membership */
        }

    replace all the messages of UNIFORM(<P; F; c>) with one synthetic message μ ≡ <P; F; c>,
        such that μ causally follows all the messages in CCONE(<P; F; c>) ;
    Put μ in pendingQ  and mark it decision ;
    DELIVER() ;
```

Figure 7: Prepare a membership change for delivery

```
procedure DELIVER():

    Loop until exit {
        Let C be:
            the set of messages in pendingQ, that do not causally follow
            any message in delayQ or pendingQ ;

         /* Note: the order of if's below is important. */
        if there are install membership messages
            choose a message μ among them deterministically ;
            pop μ from pendingQ ; M–install_p(μ) ; Memb = μ ;
        else if there is a scatter membership message μ ∈ C
            pop μ from pendingQ ; M–scatter_p(μ) ;
        else if there is a reject membership message μ ∈ C
            pop μ from pendingQ ;
        else if there is a regular message r ∈ C
            pop r from pendingQ ; M–deliver_p(r) ;
        else exit the loop ;
    }
```

Figure 8: Handle the delivery of all messages

4.4 Examples

This section provides a few example scenarios of the membership protocol. For simplicity, all the examples show membership messages only, without regular messages. We denote by capital letters the names of machines, *e.g.* A, B, C. We denote by <M;i> the i'th message from machine M. Arrows express the causal order.

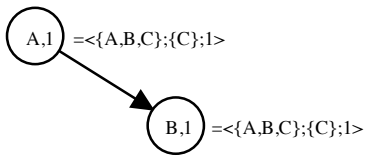

Figure 9: Simple failure detection

Figure 9: This scenario exemplifies a simple failure detection. At the start of this scenario, machines A, B, and C all have $Memb = \{A, B, C\}$, $context = 1$. The following events occur:

- A suspects C, and multicasts <A;1> $\equiv$ < $\{A, B, C\}$;$\{C\}$;*1* > . Both A and B insert <A;1> into $\mathcal{MQ}^A$ when it is $\mathcal{C}$–delivered.
- B responds with <B;1> $\equiv$ < $\{A, B, C\}$;$\{C\}$;*1* > . Both A and B insert <B;1> into $\mathcal{MQ}^B$ when it is $\mathcal{C}$–delivered.
- Now, both A and B install < $\{A, B, C\}$;$\{C\}$;*1* >, *i.e.* set $Memb$ to $\{A, B\}$, and increment $context$ to 2.

Figure 10: This scenario exemplifies three-way joining. Initially, machines A, B, and C form the singleton sets $\{A\}$, $\{B\}$, and $\{C\}$, respectively. The machines periodically use the hardware broadcast to perform "relatives search". As a result, they may intercept "foreign" messages.

- machine B intercepts a foreign message from A. It multicasts <B;1> $\equiv$ < $\{A, B\}$;$\{\}$;*1* > .
- At approximately the same time, machine A intercepts a foreign message from C, and multicasts a causally-concurrent message <A;1> $\equiv$ < $\{A, C\}$;$\{\}$;*1* > .
- C intercepts to <B;1> and responds with <C;1> $\equiv$ < $\{A, B, C\}$;$\{\}$;*1* > .
- B responds to <A;1> with <B;2> $\equiv$ < $\{A, B, C\}$;$\{\}$;*1* > .
- A responds to <C;2> with <A;2> $\equiv$ < $\{A, B, C\}$;$\{\}$;*1* > .
- Finally, A responds to <B;2> with <A;2> $\equiv$ < $\{A, B, C\}$;$\{\}$;*1* > .

Note that messages <A;1> and <A;2> (similarly, <B;1>, <B;2>) have the same context, since there are no installation events in the interim. Let us follow the internal data structures at machine A:

1. A inserts <A;1> into $\mathcal{MQ}^A$.

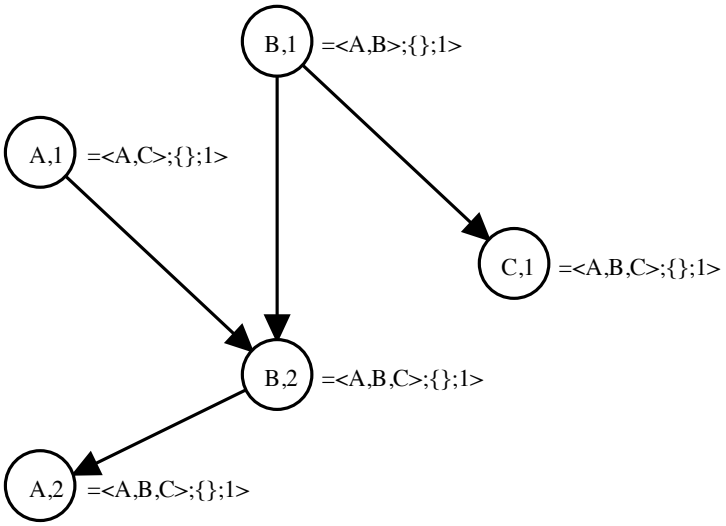

Figure 10: Three-way joining

2. A inserts <B;1>, <C;1>, <B;2> to $\mathcal{MQ}^B$, $\mathcal{MQ}^C$ respectively, without being able to unblock <A;1>.

3. A inserts <A;2> to $\mathcal{MQ}^A$. A decides to *install* <A;2> (*i.e.* install the membership set $\{A, B, C\}$), and to *reject* <A;1>.

Figure 11: This scenario exemplifies failures during joining, and the motivation for detecting hidden agreement. In this scenario, machines A and B send the messages <A;1>, <B;1>,<B;2>,<A;2>, as in the previous scenario. Next, C detaches from A and from B, such that, whether C sends messages or not, A and B receive no messages from C. The sequence of events proceeds as follows:

- A incurs the event $\mathcal{C}$–suspect$_A(C)$, and consequently sends <A;3> $\equiv$ < $\{A, B, C\}$;$\{C\}$;*1* > .
- B responds with <B;3> $\equiv$ < $\{A, B, C\}$;$\{C\}$;*1* > .

We follow the execution of the protocol at A:

1. A inserts the membership messages <A;1>, <A;2> and <A;3> into $\mathcal{MQ}^A$, and <B;1>, <B;2> into $\mathcal{MQ}^B$. Yet, A is unable to unblock the first membership message in $\mathcal{MQ}^A$.

2. <B;3> is inserted into $\mathcal{MQ}^B$.

3. A observes that <A;3> has *full agreement* (every machine in $\{A, B\}$ has sent < $\{A, B, C\}$;$\{C\}$;*1* >).

4. A decides that <A;2> has *hidden agreement* (every machine in $\{A, B, C\} \cap \{A, B\}$ has sent < $\{A, B, C\}$;$\{\}$;*1* >).

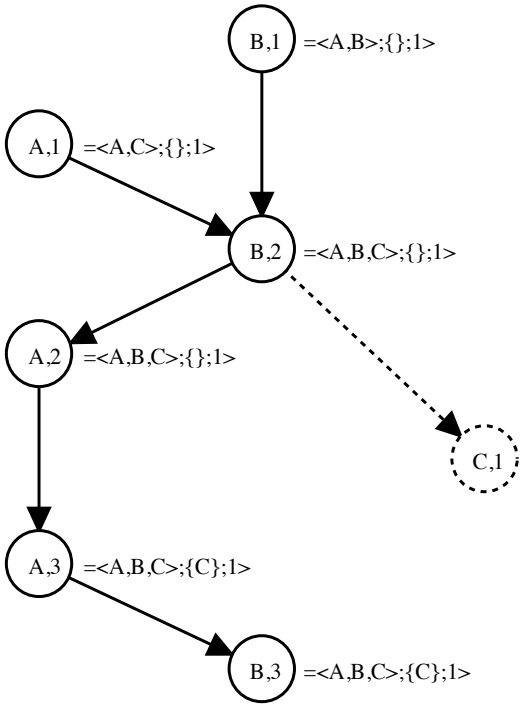

Figure 11: C disconnects during joining

5. A delivers **<**A;2**>** as a *scattered* membership, and installs **<**A;3**>**. The scattered membership indicates to the user that at machine C, the intermediate membership $\{A, B, C\}$ *may have been installed*, as indicated by the dashed message in Figure 11.

5 Correctness Proof

This section proves that the membership protocol maintains Requisites M.1, M.2, M.3, M.4, M.5. For a certain machine p, We denote its local data structures by AM_p, AF_p, $context_p$, $Memb_p$, $\mathcal{MQ}_p^*$, and the like. We omit the subscripts when they are obvious from the context.

Definition 5.1 We define an order relation $\prec$ on membership messages as follows: Let $\mu_1 \equiv$ <P_1; F_1; c_1>, $\mu_2 \equiv$ <P_2; F_2; c_2> be two membership messages. Then $\mu_1 \prec \mu_2$ iff:

- $c_1 = c_2$, $\mathrm{P}_1 \subseteq \mathrm{P}_2$ and $\mathrm{F}_1 \subseteq \mathrm{F}_2$, and at least one of the set inclusions is strong, or
- $c_1 < c_2$.

We denote $\mu_1 \preceq \mu_2$ if either $\mu_1 \prec \mu_2$, or $\mu_1 = \textsc{equiv}(\mu_2)$.

According to the protocol, the membership messages that each machine sends throughout the history are totally ordered by $\prec$. The first lemma states this fact.

Lemma 5.1 *Let μ_1 and μ_2 be two different membership messages*[3] *sent by a machine p. Then* $\mu_1 \stackrel{cause}{\longrightarrow} \mu_2 \implies \mu_1 \prec \mu_2$.

Proof: According to the protocol, at each machine, the *context* field monotonically increases. A machine may send several membership messages with the same context field; however, as long as the context does not change, AM and AF do not decrease. Therefore, the membership messages emitted by each machine are totally ordered by $\prec$. □

By Lemma 5.1, every pair of membership messages that are sent by the same machine are ordered according to their contents. *Therefore, if another machine sends the same pair of messages, it must send them in the same order.*

Lemma 5.2 *Let* $\textsc{equiv}_p(\mu_1)$, $\textsc{equiv}_p(\mu_2)$, *be a pair of different membership messages sent by p, and* $\textsc{equiv}_q(\mu_1)$, $\textsc{equiv}_q(\mu_2)$ *be a similar pair sent by q. Then*
$\textsc{equiv}_p(\mu_1) \stackrel{cause}{\longrightarrow} \textsc{equiv}_p(\mu_2) \implies \textsc{equiv}_q(\mu_1) \stackrel{cause}{\longrightarrow} \textsc{equiv}_q(\mu_2)$.

Proof: If $\textsc{equiv}_p(\mu_1) \stackrel{cause}{\longrightarrow} \textsc{equiv}_p(\mu_2)$, then, by Lemma 5.1, $\textsc{equiv}_p(\mu_1) \prec \textsc{equiv}_p(\mu_2)$. Therefore, $\textsc{equiv}_q(\mu_1) \prec \textsc{equiv}_q(\mu_2)$, and by Lemma 5.1, q must have sent them in an order that preserves $\prec$. □

The following lemma proves that membership messages that have full–agreement are promptly installed.

Lemma 5.3 *Every call to* UNBLOCK-MEMBERSHIP*(μ,* install*) incurs an $\mathcal{M}$–install event within* DELIVER*().*

[3] Note that the same membership message may be repeated by p.

Proof: In procedure UNBLOCK-MEMBERSHIP, all the messages in CCONE(μ) are removed from *delay*$\mathcal{Q}$ (they are either discarded or moved to *pending*$\mathcal{Q}$). Therefore, all the causal predecessors of UNIFORM(μ) in *pending*$\mathcal{Q}$ (if any) are delivered within the loop in DELIVER(). Thereafter, μ incurs an $\mathcal{M}$–install event. □

The lemma and theorem below show that the installation of memberships is done in consistent order.

Lemma 5.4 *Let $\mathcal{M}$–install$_p(\mu_1)$, $\mathcal{M}$–install$_p(\mu_2)$ be two events at p. Then* EQUIV$_p(\mu_1) \stackrel{cause}{\longrightarrow}$ EQUIV$_p(\mu_2)$ *if, and only if,* $\mathcal{M}$–install$_p(\mu_1) \stackrel{H}{\longrightarrow} \mathcal{M}$–install$_p(\mu_2)$.

Proof: By the definition of $\stackrel{cause}{\longrightarrow}$ and Requisite C.1, either EQUIV$_p(\mu_1) \stackrel{cause}{\longrightarrow}$ EQUIV$_p(\mu_2)$ or EQUIV$_p(\mu_2) \stackrel{cause}{\longrightarrow}$ EQUIV$_p(\mu_1)$, but not both. Thus it suffices to show that if EQUIV$_p(\mu_1) \stackrel{cause}{\longrightarrow}$ EQUIV$_p(\mu_2)$, then $\mathcal{M}$–install$_p(\mu_1) \stackrel{H}{\longrightarrow} \mathcal{M}$–install$_p(\mu_2)$. This stems from the fact that when p unblocks EQUIV$_p(\mu_2)$, it either rejects or scatters every preceding membership message that has not been unblocked yet. Therefore, since we assumed that μ_1 is installed by p, EQUIV$_p(\mu_1)$ must be unblocked (in HANDLE-MEMBERSHIP) before EQUIV$_p(\mu_2)$. Therefore, μ_1 is marked *install* before μ_2, and by Lemma 5.3, $\mathcal{M}$–install$_p(\mu_1)$ must precede $\mathcal{M}$–install$_p(\mu_2)$. □

Theorem 5.5 *The membership protocol maintains Requisite M.1.*

Proof: Let μ_1 and μ_2 be two membership messages installed by p and by q, and $\mathcal{M}$–install$_p(\mu_1) \stackrel{H}{\longrightarrow} \mathcal{M}$–install$_p(\mu_2)$. We show that $\mathcal{M}$–install$_q(\mu_1) \stackrel{H}{\longrightarrow} \mathcal{M}$–install$_q(\mu_2)$.

There exists a pair of different messages EQUIV$_p(\mu_1)$, EQUIV$_p(\mu_2)$, sent by p, since p installed μ_1, μ_2. Similarly, there exists EQUIV$_q(\mu_1)$, EQUIV$_q(\mu_2)$, sent by q, since q installed μ_1, μ_2. By Lemma 5.4, EQUIV$_p(\mu_1) \stackrel{cause}{\longrightarrow}$ EQUIV$_p(\mu_2)$. Therefore, by Lemma 5.2, EQUIV$_q(\mu_1) \stackrel{cause}{\longrightarrow}$ EQUIV$_q(\mu_2)$. Applying Lemma 5.4 again, we conclude that $\mathcal{M}$–install$_q(\mu_1) \stackrel{H}{\longrightarrow} \mathcal{M}$–install$_q(\mu_2)$. □

The membership protocol promptly responds to each received membership message. Therefore, when p sends a membership message, the communication layer guarantees that within a finite time, either p receives a "relevant" response from each member, or it gets a suspicion–notification. The following lemma states this property, which is crucial to the liveness of the protocol.

Lemma 5.6 *Let p send a membership message* $\mu_1 \equiv$ <P$_1$; F$_1$; c_1> *and let* $q \in \overline{\mu_1}$. *Assume that p does not crash. Then within a finite time, one of the following events occurs:*

1. *p $\mathcal{C}$–delivers a message μ_2 from q, such that $\mu_1 \stackrel{cause}{\longrightarrow} \mu_2$, μ_2 is the first message from q that causally follows μ_1, and one of the following cases hold:*
   - $\mu_2 \equiv$ EQUIV$_q(\mu_1)$.
   - $\mu_1 \prec \mu_2$.

- *q has sent a preceding membership message* $\mu_2' \stackrel{cause}{\longrightarrow} \mu_2$, *such that* $\mu_1 \preceq \mu_2'$.
- $\mu_2 \equiv$ <P; F; *c*> *such that* $p \in$ F.

2. *p incurs an event* $\mathcal{C}$*–suspect*$_p(q)$ *following* μ_1 .

Proof: According to Assumption C.2, either p $\mathcal{C}$–delivers a message μ_2 from q within a finite time, or it incurs the event $\mathcal{C}$–suspect$_p(q)$. According to the protocol, a message from q that immediately follows $\mathcal{C}$–delivery of μ_1 must satisfy one of the listed cases. □

We assume that the universe of machines is finite. The following lemma shows that each pending membership message is unblocked within a finite amount of time.

Lemma 5.7 *Let* π *be the first message in* $\mathcal{MQ}^p$. *Then* p *unblocks* π *within a finite time, or* p *crashes. Furthermore, if* p *does not crash, then some membership message causally following* π *is installed within a finite time.*

Proof: We consider the history segment $\mathcal{T}$ (possibly infinite), starting with the event in which π becomes the first message in $\mathcal{MQ}^p$ (*i.e.* $\pi \equiv \mathcal{MQ}^p[1]$), and continuing for as long as there are no $\mathcal{M}$–install events at p. If p does not crash, and $\mathcal{T}$ ends within a finite time, then by definition $\mathcal{T}$ is immediately followed by a $\mathcal{M}$–install event. Furthermore, since this install event occurs after π becomes the first message in $\mathcal{MQ}^p$, then it installs π or some membership message causally following π. Thus, it is sufficient to show that either p crashes, or $\mathcal{T}$ terminates within a finite time.

The membership messages that p sends during $\mathcal{T}$ contain monotonically increasing P, F sets (in the obvious sense of inclusion). According to protocol, $context_p$ also monotonically increases; however, $context_p$ actually increases only if there is an install event. Therefore, since we assume that the universe of machines is finite, there exists a **maximal** membership message μ sent by p during its history up to, and including, $\mathcal{T}$[4].

By assumption C.2, within a finite time after μ is sent, one of the following events occur:

1. p crashes ($crash_p \in H$). This satisfies one of the lemma's alternatives.

2. p suspects some $q \in \overline{\mu}$, *i.e.* p incurs an event $\mathcal{C}$–suspect$_p(q)$. If $\mathcal{C}$–suspect$_p(q)$ had occurred within $\mathcal{T}$, then p would have sent within $\mathcal{T}$ a new membership message that includes q in the F set. This would contradict the maximality of μ within $\mathcal{T}$. Therefore, $\mathcal{T}$ ends before the event $\mathcal{C}$–suspect$_p(q)$, and therefore $\mathcal{T}$ is finite.

3. For every $q \in \overline{\mu}$, p $\mathcal{C}$–delivers a message μ_q from q, such that $\mu \stackrel{cause}{\longrightarrow} \mu_q$. Combining the possible cases of Lemma 5.6 for all q's, we obtain three possibilities:

   (a) There exists $q \in \overline{\mu}$ that has sent a membership message μ_q, such that $\mu_q' \stackrel{cause}{\longrightarrow} \mu_q$ and $\mu \prec \mu_q'$ (note that $\stackrel{cause}{\longrightarrow}$ is reflexive, so possibly $\mu_q' = \mu_q$). In response to μ_q', p must send a membership message μ', such that $\mu \prec \mu'$. By the maximality of μ, μ' is sent after the period $\mathcal{T}$ ends. This proves that $\mathcal{T}$ ends within a finite time.

[4] $\mathcal{T}$ itself might not contain any multicast events at p. In this case, μ may be π or some membership message following π, sent before $\mathcal{T}$ starts

(b) There exists $q \in \overline{\mu}$, such that the F set in μ_q contains p. In response, p *reverses* the suspicion. As in case 2 above, this leads to the conclusion that $\mathcal{T}$ must be finite.

(c) For every $q \in \overline{\mu}$, q sent $\mu'_q \equiv \text{EQUIV}_q(\mu)$, and $\mu'_q \stackrel{cause}{\longrightarrow} \mu_q$. At latest, when p has $\mathcal{C}$–delivered all of these μ_q's, μ is unblocked. If μ is unblocked, then a certain membership message that causally follows μ can be installed (and will be installed within a finite time, by Lemma 5.3). Therefore, within a finite time after μ is sent, p incurs an install event, that terminates $\mathcal{T}$. □

Fault–suspicions are "stable": until a suspected machine is removed from the membership, it cannot rejoin it. The lemma below states this property:

Lemma 5.8 *Let* $\mu_1 \equiv$ <$\mathrm{P}_1; \mathrm{F}_1; c_1$>, $\mu_2 \equiv$ <$\mathrm{P}_2; \mathrm{F}_2; c_2$> *be membership messages sent by p, such that:* $\mu_1 \stackrel{cause}{\longrightarrow} \mu_2$, *and* $\mathcal{C}$*–multicast*$_p(\mu_2)$ *occurs before* μ_1 *is unblocked. Then* $F_2 \supseteq F_1$.

Proof: This claim results from the rule that for every q, as long as there are any pending membership messages that contain q in their F set, $q \in AF_p$. When μ_2 is sent, the faults set F_2 contains AF_p, which, since μ_1 is still pending, includes F_1. □

The following lemma and resulting theorem prove Requisite M.2. This shows that suspected machines are removed from the membership within a finite time. It stems from (a) the fact that suspicions are stable, and (b) as proved in the previous lemma, the membership protocol is nonblocking.

Lemma 5.9 *Let p incur an event* $\mathcal{C}$*–suspect*$_p(q)$. *Assume that p does not crash. Then within a finite time after this event, p installs a membership* μ, *such that* $q \notin \overline{\mu}$.

Proof: Following $\mathcal{C}$–suspect$_p(q)$, p appends q to AF_p and sends a membership message $\mu =$ <$AM; AF; c$> . Let μ' be a membership message following μ, that is sent before μ is unblocked. Then, by lemma 5.8, μ' contains q in its faults set. Thus, every membership message sent by p after μ and before μ is unblocked contains q is its faults–set.

By Lemma 5.7, every membership message that precedes μ in $\mathcal{MQ}^p$ is removed (unblocked) within a finite time. Thus, within a finite time, μ becomes the first message in $\mathcal{MQ}^p$. Applying Lemma 5.7 again, we obtain that μ is unblocked within a finite time. Furthermore, this means that a membership message that causally follows μ in $\mathcal{MQ}^p$ is installed, and as we showed above, the installed membership message contains q in its F set. □

Theorem 5.10 *The membership protocol maintains Requisite M.2.*

Proof: This follows directly from Assumption C.2 and Lemma 5.9. □

Next, we prove Non–triviality.

Theorem 5.11 *The membership protocol maintains Requisite M.3.*

Proof: Let $T = \{t_1, \ldots, t_n\}$ be a set of machines. We need to show that there exists a history in which T is an installed membership. We provide a proof–sketch only, since a complete proof will be quite cumbersome, and we believe that the intuition behind it suffices here.

By assumption C.3, there exists a history H in which the following sequence of messages is successfully exchanged, such that no suspicions and no crashes occur:

$$\begin{array}{lll} < & t_1 \ , & \langle T; \emptyset; 1 \rangle \ > \\ < & t_2 \ , & \langle T; \emptyset; 1 \rangle \ > \\ \ldots & & \\ < & t_n \ , & \langle T; \emptyset; 1 \rangle \ > \end{array}$$

Under the above assumptions, every machine in T installs the membership T following this exchange of messages. □

The following lemma proves that the membership module preserves causality.

Lemma 5.12 *If $m \xrightarrow{cause} m'$ are two messages, such that $\mathcal{M}$–deliver$_p(m)$ and $\mathcal{M}$–deliver$_p(m')$ are events of H, then $\mathcal{M}$–deliver$_p(m) \xrightarrow{H} \mathcal{M}$–deliver$_p(m')$.*

Proof: From assumption C.1 of the causal communication layer, $\mathcal{C}$–deliver$_p(m) \xrightarrow{H} \mathcal{C}$–deliver$_p(m')$. Therefore, m is inserted into either $delay\mathcal{Q}_p$ or $pending\mathcal{Q}_p$ before m'. In DELIVER(), m' becomes deliverable only after m is removed from these queues. Since we assumed that m was not discarded, m must be delivered before m'. □

The following lemma proves that messages are $\mathcal{M}$–delivered only from members of the current membership.

Lemma 5.13 *Let m be a regular message, and let $\mathcal{M}$–deliver$_p(m, s)$ be an event of p, such that $\mathcal{M}$–install$_p(\mu)$ is the last install event at p preceding $\mathcal{M}$–deliver$_p(m, s)$. Then $s \in \overline{\mu}$.*

Proof: Assume that $\mathcal{M}$–install$_p(\mu)$ is the last install event at p preceding $\mathcal{M}$–deliver$_p(m, s)$. If $\mathcal{M}$–install$_p(\mu) \xrightarrow{H} \mathcal{C}$–deliver$_p(m, s)$, then when m is $\mathcal{C}$–delivered, $Memb_p = \overline{\mu}$. In this case, unless $s \in \overline{\mu}$, message m is delayed in $delay\mathcal{Q}$ until after a new membership is installed.

Otherwise, $\mathcal{C}$–deliver$_p(m, s) \xrightarrow{H} \mathcal{M}$–install$_p(\mu)$. By assumption, m is delivered after μ, and therefore, m is *delayed* at p until after μ is unblocked. There may be two reasons for delaying m:

case 1: Message m was inserted into $delay\mathcal{Q}$ upon arrival, and was moved into $pending\mathcal{Q}$ only when μ was unblocked. Before μ is unblocked, any message in CCONE(μ) is either delivered or discarded by p. Therefore, $m \notin$ CCONE(μ), and procedure UNBLOCK-MEMBERSHIP moves m into $pending\mathcal{Q}$ only if $s \in \overline{\mu}$.

case 2: Message m was inserted into $pending\mathcal{Q}$ before μ was unblocked, and was delayed from delivery because m causally follows some message in UNIFORM(μ). This means that when m was inserted into $pending\mathcal{Q}$ at p, s has belonged to the installed membership. The message

m remains in $pending\mathcal{Q}$ as long as no membership that removes s is installed (and until m is delivered). In particular, if m remains after μ is installed, $\overline{\mu}$ must (still) include s.

□

The following three lemmata prove that between two consecutive membership changes, the same set of messages are delivered everywhere. This proves the second clause of the virtual synchrony requisite.

Lemma 5.14 *Let $\mathcal{M}$–install$_p(\mu)$, $\mathcal{M}$–install$_p(\mu')$, be consecutive install events at p. Then any message m such that:*

$$\mathcal{M}\text{–install}_p(\mu) \xrightarrow{H} \mathcal{M}\text{–deliver}_p(m) \xrightarrow{H} \mathcal{M}\text{–install}_p(\mu')$$

satisfies: $m \in \left(\text{CCONE}(\mu') \setminus \text{CCONE}(\mu)\right)$ *and the sender of m is in* $\overline{\mu}$.

Proof:
Let m be a message such that $\mathcal{M}$–install$_p(\mu) \xrightarrow{H} \mathcal{M}$–deliver$_p(m) \xrightarrow{H} \mathcal{M}$–install$_p(\mu')$.

Denote the sender of m by s. By assumption, $\mathcal{M}$–install$_p(\mu)$, $\mathcal{M}$–install$_p(\mu')$ are consecutive, and therefore $\mathcal{M}$–install$_p(\mu)$ is the last install event preceding $\mathcal{M}$–deliver$_p(m)$. From Lemma 5.13, s must be in $\overline{\mu}$.

Since p installs μ, any message in CCONE(μ) is either $\mathcal{M}$–delivered or discarded by p before $\mathcal{M}$–install$_p(\mu)$. It is left to show that m is in CCONE(μ').

Assume to the contrary that $m \notin$ CCONE(μ'). The following facts hold:

F–1 m is concurrent to UNIFORM(μ').
This stems from two facts: First, we assumed that m is not in CCONE(μ'), so it does not causally precede any message in UNIFORM(μ'). Second, m does not causally follow any message in UNIFORM(μ'). The reason is that a message that causally follows any message in UNIFORM(μ') cannot be delivered until the preceding message is unblocked. This unblocking occurs only once UNIFORM(μ') has full agreement. Then, UNIFORM(μ') is replaced with a synthetic membership message that follows CCONE(μ'), and procedure DELIVER delivers this membership message before any messages outside CCONE(μ'). This shows that a message that causally follows any message in UNIFORM(μ'), and is not in CCONE(μ'), must be delivered after $\mathcal{M}$–install$_p(\mu')$. Since this contradicts our assumption, m does not follow any message in UNIFORM(μ').

F–2 Denote $\mu' \equiv$ <P′; F′; c'> . Then $s \in$ F′ .
From fact F–1, s cannot be any machine in $\overline{\mu}'$. Thus $s \in \overline{\mu} \setminus \overline{\mu}'$. Since μ and μ' incur consecutive install events at p, the faults set F′ in μ' must contain the difference–set $(\overline{\mu} \setminus \overline{\mu}')$.

F–3 Between the events $\mathcal{C}$–deliver$_p(m)$ and $\mathcal{M}$–install$_p(\mu')$, $s \in AF_p$.
Since $m \notin$ CCONE(μ'), m is $\mathcal{C}$–delivered at p after p has sent EQUIV$_p(\mu')$ (and before μ' is installed). According to the protocol, as long as μ' is not unblocked, AF_p contains F′. By Fact F–2, during this period, $s \in AF_p$.

By fact F–3, upon $\mathcal{C}$–deliver$_p(m)$, m must be inserted into $delay\mathcal{Q}$. Message m cannot be moved from $delay\mathcal{Q}$ as long as $s \in AF_p$, or, by fact F–3, at least until μ' is unblocked.

When μ' is unblocked, procedure UNBLOCK-MEMBERSHIP discards m, because $s \in \mathrm{F}'$, and $m \notin$ CCONE(μ'). This contradicts the assumption that m is $\mathcal{M}$–delivered by p . □

Lemma 5.15 *Let $\mathcal{M}$–install$_p(\mu)$, $\mathcal{M}$–install$_p(\mu')$, be consecutive install events at p. Let m be a message sent by a machine in $\overline{\mu}$, such that $m \in \Big(\text{CCONE}(\mu') \setminus \text{CCONE}(\mu)\Big)$. Then*

$$\mathcal{M}\text{–install}_p(\mu) \xrightarrow{H} \mathcal{M}\text{–deliver}_p(m) \xrightarrow{H} \mathcal{M}\text{–install}_p(\mu') \ .$$

Proof: By Lemma 5.14, if m is in fact delivered, then it must be delivered between the indicated pair of membership installations. We need to show that m is indeed delivered (*i.e.* not discarded, or somehow delayed forever).

First, we show that m is not discarded by any instance of the procedure UNBLOCK-MEMBERSHIP up to, and including, UNBLOCK-MEMBERSHIP(μ). Assume to the contrary, that a preceding (or equal) instance UNBLOCK-MEMBERSHIP(μ'') discards m. From this assumption, we can deduce the following:

1. Since μ'' is installed before (or is) μ, then by Lemma 5.4, EQUIV$_p(\mu'') \xrightarrow{cause}$ EQUIV$_p(\mu)$ [5].
2. Whence, since $m \notin$ CCONE(μ), then also $m \notin$ CCONE(μ'') .
3. Denote the sender of m by s. By hypothesis s is in $\overline{\mu}$. As $m \notin$ CCONE(μ''), the reason that procedure UNBLOCK-MEMBERSHIP(μ'') discards m must be that s is in the faults–set of μ''. This means that $s \notin \overline{\mu''}$, and therefore $\mu'' \neq \mu$.
4. We know that m is $\mathcal{C}$–delivered at p before μ'' is installed. Additionally, since $m \notin$ CCONE(μ), m is $\mathcal{C}$–delivered after EQUIV$_p(\mu)$ is sent. Therefore, together with item 3 above, we have

   $$\mathcal{C}\text{–multicast}_p(\text{EQUIV}_p(\mu'')) \xrightarrow{H} \mathcal{C}\text{–multicast}_p(\text{EQUIV}_p(\mu)) \xrightarrow{H} \mathcal{M}\text{–install}_p(\mu'') \ .$$

   By Lemma 5.8, this implies that the faults–set of μ contains s. This contradicts the fact that $s \in \overline{\mu}$.

We proceed to show that UNBLOCK-MEMBERSHIP(EQUIV$_p(\mu')$, *install*) delivers m. Message m is $\mathcal{C}$–delivered before μ' is installed, because $m \in$ CCONE(μ'). If m is $\mathcal{M}$–delivered before the unblocking, then we are done. Otherwise, procedure UNBLOCK-MEMBERSHIP moves from $delay\mathcal{Q}$ to $pending\mathcal{Q}$ every message in CCONE(μ') (that is not already delivered), that is from a sender in $\overline{\mu}$. Therefore, it moves m to $pending\mathcal{Q}$. The procedure also replaces UNIFORM(μ') with a synthetic membership message μ' that causally follows all the messages in CCONE(EQUIV$_p(\mu')$). Therefore, In procedure DELIVER, the event $\mathcal{M}$–install$_p(\mu')$ must follow the delivery of m (thereby proving that m cannot be "stuck" in $pending\mathcal{Q}$ somehow). This completes the proof that m is indeed delivered. □

[5] Recall that $\xrightarrow{cause}$ is reflexive.

Lemma 5.16 *Let $\mathcal{M}$–install$_p(\mu)$, $\mathcal{M}$–install$_p(\mu')$ be consecutive install events at p, and let $\mathcal{M}$–install$_q(\mu)$, $\mathcal{M}$–install$_q(\mu')$ be events of q, in H. Then*

$$\begin{aligned}\forall m: \quad & \mathcal{M}\text{–install}_p(\mu) \xrightarrow{H} \mathcal{M}\text{–deliver}_p(m) \xrightarrow{H} \mathcal{M}\text{–install}_p(\mu') \Rightarrow \\ & \mathcal{M}\text{–install}_q(\mu) \xrightarrow{H} \mathcal{M}\text{–deliver}_q(m) \xrightarrow{H} \mathcal{M}\text{–install}_q(\mu') \ .\end{aligned}$$

Proof: If $\mathcal{M}$–install$_q(\mu)$ and $\mathcal{M}$–install$_q(\mu')$ are consecutive install events at q, then by Lemmata 5.14, 5.15, the set of messages delivered by both p and q between μ and μ' is exactly equal to the set of messages $m \in (\text{CCONE}(\mu') \setminus \text{CCONE}(\mu))$, such that m is from a sender in $\overline{\mu}$. Thus it suffices to show that the install events are consecutive at q.

By Theorem 5.5, $\mathcal{M}$–install$_q(\mu) \xrightarrow{H} \mathcal{M}$–install$_q(\mu')$. Suppose there is some intervening install event at q. This event cannot take place at p because μ and μ' are consecutive there. But then, there are two possibilities:

1. If $\mathcal{M}$–install$_q(\mu'')$ takes place before q sends EQUIV$_q(\mu')$, then EQUIV$_q(\mu')$ will have a wrong context for p; Procedure MERGE-MEMBERSHIP guarantees that full agreement for μ' cannot be obtained at p, so $\mathcal{M}$–install$_p(\mu')$ cannot take place.

2. Otherwise, EQUIV$_q(\mu')$ and EQUIV$_q(\mu'')$ must have co-resided in $\mathcal{MQ}_q^q$. Note that, $p \in \overline{\mu''}$, or else, between μ and μ', p was suspected by q, and thus p would have first reversed the suspicion. Furthermore, because μ', μ'' coreside at $\mathcal{MQ}^q$, UNIFORM$(\mu'') \subseteq$ CCONE(μ'). Therefore, p must have also received full agreement for μ'' before it installed μ', and would have installed μ'' first.

This contradiction completes the proof of the lemma. □

The following theorem states that the protocol maintains all the virtual–synchrony requisites.

Theorem 5.17 *The membership protocol maintains Requisite M.4.*

Proof: Follows from Lemma 5.12, Lemma 5.16, and Lemma 5.13. □

The following theorem proves that an installed membership represents an obligation by all the members, in the sense of Requisite M.5.

Theorem 5.18 *The membership protocol maintains Requisite M.5.*

Proof: Let $\mathcal{M}$–install$_p(\mu)$ be an event of H. Let q be a machine in $\overline{\mu}$. Assume that q does not crash in H. Since p has installed μ, q has sent EQUIV$_q(\mu)$. We will show that q cannot *reject* EQUIV$_q(\mu)$. This will prove that (since q does not crash) either $\mathcal{M}$–install$_q(\mu) \in H$, or $\mathcal{M}$–scatter$_q(\mu) \in H$.

Let us observe the execution of procedure HANDLE-MEMBERSHIP at q when EQUIV$_q(\mu)$ is at the head of $\mathcal{MQ}^q$. There may be several such instances of HANDLE-MEMBERSHIP, the last of which unblocks EQUIV$_q(\mu)$, by Lemma 5.7. We will focus on that last instance.

Let k be the minimal index found at this instance of HANDLE-MEMBERSHIP, such that $\mathcal{MQ}^q[k]$ has full agreement. If k is 1, then q installs $\text{EQUIV}_q(\mu) \equiv \mathcal{MQ}^q[1]$, and we are done.

Otherwise, $\mathcal{MQ}^q[1]$ and $\mathcal{MQ}^q[k]$ are different messages, and $\mathcal{MQ}^q[1] \xrightarrow{cause} \mathcal{MQ}^q[k]$. Every machine $r \in \overline{\mathcal{MQ}^q[1]} \cap \overline{\mathcal{MQ}^q[k]}$ satisfies the following:

- r sent a message $\text{EQUIV}_r(\mathcal{MQ}^q[1])$ (because p has full agreement on $\mu = \text{EQUIV}_p(\mathcal{MQ}^q[1])$) .
- r sent a message $\text{EQUIV}_r(\mathcal{MQ}^q[k])$ (since q has full agreement on it).
- By Lemma 5.2, $\text{EQUIV}_r(\mathcal{MQ}^q[1]) \xrightarrow{cause} \text{EQUIV}_r(\mathcal{MQ}^q[k])$.
- Therefore, $\text{EQUIV}_r(\mathcal{MQ}^q[1]) \in \text{CCONE}(\mathcal{MQ}^q[k])$.

Thus, $\mathcal{MQ}^q[1]$ has hidden agreement, and according to the protocol, q *scatters* $\mu \equiv \mathcal{MQ}^q[1]$ in this case. □

6 Conclusions

This paper has done the following: It provided a full definition of the membership problem in asynchronous distributed communication systems, and presented a full solution that operates above a causal communication layer. The definition allows multiple partitions to exist.

In a world of growing dependency on computers, the ability to continue operation in a dynamic environment is crucial. Algorithms that depend on the existence of a *primary partition* in the system are not sufficient to meet all the needs of distributed applications. Therefore, we have assumed from the outset that the network might partition, and sought the semantics that provide the user with accurate information within each partition. The importance of this approach is twofold: First, it defines consistent semantics for operation with multiple partitions (which is relevant even in the case that **eventually**, all but one of the partitions are forced to give up). Second, it allows for continued partitionable operation, which is useful in various applications.

The membership protocol presented here shows that the reconfiguring–operation need not stop the flow of messages in the system. Cascading reconfigurations can be handled in a pipeline of agreements. The utilization of broadcast communication makes the protocol especially efficient within local clusters.

The principles that underlie our approach have been proved useful in several environments ([3, 4, 29]):

- Increasing both the joining–set and faults–set until a decision is made (and thus making suspicions stable) guarantees termination within a finite time.
- Handling multiple configuration changes together guarantees the virtual synchrony principle.
- Allowing partitionable operation increases the flexibility of the membership service.
- The symmetrical joining avoids serializing joining, and allows merging of full sets.
- Utilizing broadcast communication for efficiency within local clusters.

The problem definition presented here gives the basis for consistent operation with multiple partitions. One of the directions set forth by this research is the development of additional communication layers, to support specific applications. For example, the work of Moser et al. [25] on Extended Virtual Synchrony binds the membership service with uniform message delivery, in order to support replicated databases in partitionable environments.

7 Acknowledgements

We have benefitted from the experience gathered in the Transis project at the Hebrew University, the Isis and the Horus projects at Cornell University, and the Totem project at the University of California, Santa Barbara, and from many discussions with the members of these groups.

References

[1] O. Amir, Y. Amir, and D. Dolev. Highly Available Application in the Transis Environment - a Test Case. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France*, June 1993. to appear as LNCS proceedings.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, pages 292–312, November 1992.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[4] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Intl. Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[5] 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.

[6] K. Birman and T. Joseph. Realiable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, February 1987.

[7] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th Ann. Symp. Operating Systems Principles*, pages 123–138. ACM, Nov 87.

[8] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, 1991.

[9] K. P. Birman. *Reliable Distributed Computing with the Isis Toolkit*, chapter Virtual Synchrony Model. IEEE Press, 1993. to appear.

[10] D. R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Trans. Comp. Syst.*, 2(3):77–107, May 1985.

[11] F. Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4(4):175–187, April 1991.

[12] S. E. Deering. Host extensions for IP multicasting. RFC 1112, SRI Network Information Center, August 1989.

[13] S. E. Deering and D. R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Trans. Comp. Syst.*, 8(2):85–110, May 1990.

[14] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[15] H. Garcia-Molina and A. Spauster. Message Ordering in a Multicast Environment. In *9th Intl. Conf. Distributed Computing Systems*, pages 354–361. IEEE, June 89.

[16] K. J. Goldman. Highly Concurrent Logically Synchronous Multicast. *Distributed Computing*, 4(4):189–208, 1991.

[17] F. Jahanian and W. Moran. Strong, Weak and Hybrid Group Membership. unpublished, IBM internal draft, 1992.

[18] F. S. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije University, 1992.

[19] M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *Intl. Conference on Distributed Computing Systems*, number 11, pages 882–891, May 1991.

[20] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[21] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 78.

[22] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. Parallel & Distributed Syst.*, (1), Jan 1990.

[23] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership Algorithms for Asynchronous Distributed Systems. In *Intl. Conf. Distributed Computing Systems*, May 91.

[24] 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.

[25] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. 1993.

[26] S. Navaratnam, S. Chanson, and G. Neufeld. Reliable Group Communication is Distributed Systems. In *Intl. Conf. Distributed Computing Systems*, pages 439–446. IEEE, June 88.

[27] 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.

[28] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *proc. annual ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.

[29] R. van Renesse, R. Cooper, B. Glade, and P. Stephenson. A Risc Approach to Process Groups. In *Proceedings of the 5th ACM SIGOPS Workshop*, pages 21–23, September 1992.

[30] P. Verissimo, L. Rodrigues, and J. Rufino. The Atomic Multicast Protocol (AMp). In D. Powell, editor, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, pages 267–294. Springer-Verlag, 1991.