Partitionable Group Membership: Specification and Algorithms

Özalp Babaoğlu  Renzo Davoli  Alberto Montresor

Technical Report UBLCS-97-1

January 1997
Revised May 1997

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)
The University of Bologna Department of Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS or via WWW at URL http://www.cs.unibo.it/. Plain-text abstracts organized by year are available in the directory ABSTRACTS. All local authors can be reached via e-mail at the address last-name@cs.unibo.it. Questions and comments should be addressed to tr-admin@cs.unibo.it.

Recent Titles from the UBLCS Technical Report Series

95-13  Optimal Multi-Block Read Schedules for Partitioned Signature Files, P. Ciaccia, August 1995.
96-5   The Shape of Shade: a Coordination System, S. Castellani, P. Ciancarini, D. Rossi, March 1996.
96-7   Using Bayesian Belief Networks for the Automated Assessment of Students’ Knowledge of Geometry Problem Solving Procedures, M. Roccetti, P. Salomoni, March 1996 (Revised March 1997).
96-9   Towards an Algebra of Actors, M. Gaspari, April 1996.
96-10  Mobile Petri Nets, A. Asperti, N. Busi, May 1996.
96-12  A Logic Coordination Language Based on the Chemical Metaphor, P. Ciancarini, D. Fogli, M. Gaspari, July 1996.
Partitionable Group Membership: Specification and Algorithms

Özalp Babaoğlu \(^1\) Renzo Davoli \(^1\) Alberto Montresor \(^1\)

Technical Report UBLCS-97-1

January 1997
Revised May 1997

Abstract

We give a formal specification for a partitionable group membership service in asynchronous distributed systems. Our specification is motivated by the requirements for building “partition-aware” applications that can continue operating without blocking in multiple concurrent partitions and reconfigure themselves dynamically when partitions merge. The specified service is sound in the sense that it guarantees liveness, excludes trivial solutions and is implementable in practical asynchronous distributed systems where certain stability conditions hold.

---

1. Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, Bologna 40127 (Italy), Tel: +39 51 354504, Fax: +39 51 354510, Email: \{babaoglu,davoli,montresor\}@CS.UniBO.IT
1 Introduction

Consider the problem of managing some network service (e.g., WWW, domain name translation, authentication, printing) that can be provided by any one of a set of servers in a given system. New servers may be added and existing ones removed at will by an administrator. The service should remain available in every partition that contains at least one server. A single active server is able to service all requests within a given partition, thus multiple active servers in the same partition should be avoided. These requirements must be satisfied in the presence of server crashes, recoveries, network partitions and merges. A solution to the problem must activate a new server if the currently active one is removed from the system, if it crashes or if it is partitioned. And when a new server is added, a crashed server recovers or when partitions merge, redundant instances of active servers in each partition should be suspended. This simple problem happens to be an instance of the class of partition-aware applications that are able to make progress in multiple concurrent partitions without blocking. In general, partitions may result in service reduction or service degradation but do not necessarily render application services completely unavailable.

A solution to this problem can be constructed based on the process group paradigm. The set of potential servers form a group and track its membership through view changes that abstract away voluntary joins and leaves, crashes, recoveries, partitions and merges. Each server applies a local decision function to its current view to determine if it should be active or passive. In order to meet the application requirements, the view that each process has of the group composition has to be constructed with care. Otherwise, some partitions may end up with no service (views include processes that are in fact not in the partition) or with multiple instances of the service (views are mutually inconsistent or exclude processes that are in fact present). The goal of a partitionable group membership service is to provide a systematic solution to these and similar problems. Informally, a partitionable group membership service tracks changes in the group’s composition due to requests to join and leave the group voluntarily, or due to partitions and merges. Ideally, there should be a common view of the group’s composition that is shared by all of its members and this view should include exactly those members that are operational. This is clearly not feasible in a partitionable system where processes in different partitions will have different views of the group membership. Nevertheless, each such view should be shared by its components and should correspond to a maximal subset of mutually reachable processes. Translating these ideas into a formal specification and showing that a service based on this specification is solvable in an asynchronous system are goals of this work.

Partitions are a fact of life in most practical distributed systems and they tend to become more frequent as the geographic extent of the system grows or its connectivity weakens due the presence of wireless links. In addition to accidental partitions caused by failures, mobile computing systems that support “disconnected operation” have to face partitions when units are deliberately unplugged from the network. What distinguishes partitions from ordinary communication failures is that they disrupt communication between sites and the usual system layers cannot hide this fact from applications. To do so requires special communication layers that buffer messages at their origin throughout a partition and retransmit them upon reconnection. Even if all partitions are eventually repaired, this approach may be impractical for several reasons. First, the number of messages that need to be buffered for retransmission during extended periods where communication is interrupted may grow arbitrarily large. Second, communication state information has to survive across site failures or power cycles, and thus has to be maintained in stable storage. But more importantly, portions of an application that span multiple partitions remain blocked until communication is restored, thus precluding continued availability in concurrent partitions.

Specifying properties for fault-tolerant distributed services in asynchronous systems requires a delicate balance between two conflicting goals. The specification must be strong enough to exclude degenerate or trivial solutions, yet it must be weak enough to be implementable [2]. Formal

2. Note that Cristian and Shmuck consider the same problem but in a timed asynchronous system [6].
specification of a partitionable group membership service in an asynchronous system has proven to be elusive and numerous prior attempts have been unsatisfactory [16, 1, 8, 9, 11, 3, 18]. Anceaume et al. discuss at length these shortcomings [2]. In summary, existing specifications admit solutions that suffer from one or all of the following problems: (i) they are informal or ambiguous [18, 3, 10], (ii) they cease to install new views even in cases where the group membership continues to change [11], (iii) they capriciously split the group into several concurrent views, possibly down to singleton sets [16, 1, 8, 9, 11], (iv) they capriciously install views without justification from the operating environment [8, 9]. The lack of a satisfactory formal specification also makes it impossible to argue the correctness of various partitionable group membership service implementations that have been proposed.

The contributions of this paper are twofold: (1) We give a formal specification for partitionable group membership services that guarantees liveness and excludes useless solutions, (2) We prove that our specification is implementable in practical asynchronous systems where certain stability conditions hold. It is impossible to argue in abstract terms the utility of any specification. Ultimately, what determines if a specification is useful or not is whether it helps in solving real problems. In a companion paper, we demonstrate the usefulness of our specified service by showing how practical partition-aware applications can indeed be easily programmed when group membership is integrated with a reliable multicast service called view synchrony [5].

2 System Model

We adopt notation and terminology that is similar to that of Chandra and Toueg [19]. For simplicity, we consider a system consisting of a single group that has already been formed and for which there are no explicit join or leave requests. In other words, in the absence of failures, composition of the group is defined by a finite, static set $I$. The composition of the group may change due to process crashes, network partitions and merges, as described below. Processes are associated unique names that they maintain throughout their life, even across partitions and merges.

A communication network implements channels connecting pairs of processes along with the primitives $send()$ and $recv()$ for sending and receiving messages over them. In the absence of failures, the network is connected and each process can communicate with every other process. The system is asynchronous in the sense that neither communication delays nor relative process speeds can be bounded. Practical distributed systems often have to be considered as being asynchronous since transient failures, unknown scheduling strategies and variable loads on the computing and communication resources make it impossible to bound delays. To simplify the presentation, we make reference to a discrete global clock whose ticks coincide with the natural numbers in some unbounded range $T$. This simplification is not in conflict with the asynchrony assumption since processes are not allowed to access the global clock.

2.1 Global Histories

The execution of a distributed algorithm results in each process taking a (possibly null) step at each clock tick. Only some of these steps are relevant with respect to group membership. Relevant steps result from the execution of events from the set $E$ which includes $send()$ and $recv()$ denoting the underlying communication primitives for sending and receiving messages. In Section 3 we extend this set with other events related to group membership. The global history of an execution is a function $\sigma$ from $I \times T$ to $E \cup \{\epsilon\}$ where $\epsilon$ denotes any non-relevant or null step. If at time $t$ process $p$ executes an event $e \in E$, then $\sigma(p,t) = e$. Otherwise, $\sigma(p,t) = \epsilon$ indicating that at time $t$, process $p$ either took no step or took a non-relevant step. Given some interval $I$ of $T$, we write $e \in \sigma(p,I)$ if $p$ executes event $e$ sometime during the interval $I$ of global history $\sigma$ (i.e., $\exists t \in I : e = \sigma(p,t)$).
Processes may fail by crashing whereby they halt prematurely. For simplicity, we do not consider process recovery after a crash. The evolution of process failures during global history $\sigma$ is captured through a crash pattern function $C_\sigma$ from $T$ to $2^I$ where $C_\sigma(t)$ denotes the set of processes that have crashed by time $t$. That is, $p \in C_\sigma(t) \iff \forall t' \geq t : \sigma(p, t') = c$. In this formula and all others that follow, free variables are assumed to be universally quantified over their respective domains. In the following, we will omit explicit references to the global history $\sigma$ when it is clear from context. Since crashed processes do not recover, we have $C(t) \subseteq C(t + 1)$. With $Correct(C) = \{ p \mid \forall t : p \not\in C(t) \}$ we denote those processes that never crash, and thus, are correct in $C$.

Communication failures may cause the network to partition and disable communication between groups of processes. Unlike process crashes which are permanent, communication failures may be temporary, whereby partitions merge after repairs and communication restored between their respective sets of processes. Intuitively, partitions correspond to situations where sets of processes are unable to communicate with other. As such, they can be defined only in the context of a specific communication service implemented by the $send()$ and $recv()$ primitives. For example, a process may appear to be partitioned from another process when it is “pinged”, but the same process may appear to be reachable when communicating through email. This is because the two communication services being considered have significantly different properties with respect to message buffering and timeout intervals before they cease retransmissions. With respect to ping-pong, even a short transient communication failure may be perceived as a partition. With respect to email, however, a communication failure has to persist for an extended period (typically 3 days) before the destination is declared partitioned.

Whereas correct/crashed are attributes of an individual process, partitions can only be defined between pairs of processes. For example, at time $t$, process $p$ may be partitioned from $q$ but not from $r$. Furthermore, partitions are not necessarily “clean” but they may result in sets of mutually-communicating components that are not disjoint. Consequently, being able to communicate is not necessarily a transitive property in partitionable systems. This is a common experience in wide-area networks, including the Internet, where it may be possible to telnet from a site $A$ to a site $B$, and telnet from $B$ to a site $C$, while it is impossible to telnet from $A$ to $C$.

With these considerations, we now give a formal definition for process pairs being reachable or partitioned during an execution. The group membership problem in partitionable systems will be specified based on this classification in the same manner that a problem such as consensus is specified based on correct/crashed classification of processes in systems that do not partition.

Let $last(p, q, t, \sigma)$ denote the last message sent from process $p$ to $q$ by time $t$ in global history $\sigma$. In other words,

$$(last(p, q, t, \sigma) = m) \iff \exists t' \leq t : \sigma(p, t') = send(m, q) \land \forall m' : send(m', q) \in \sigma(p, [t', t]).$$

If $p$ has not sent any message to $q$ by time $t$, then $last(p, q, t, \sigma) = \bot$. The evolution of communication failures and repairs during a global history $\sigma$ is captured through a partition pattern which is a function $U_\sigma$ from $I \times T$ to $2^I$ defined as follows:

$$q \in U_\sigma(p, t) \iff recv(last(p, q, t, \sigma)) \not\in \sigma(q, T) \text{ if } p \neq q \text{ and } last(p, q, t, \sigma) \neq \bot.$$
Definition 2.1 (a) Process $q$ is partitioned from $p$ at time $t$, denoted $p \not\rightarrow_t q$, if $q \in U(p, t)$.
(b) Process $q$ is reachable from $p$ at time $t$, denoted $p \sim_t q$, if $q \not\in U(p, t)$.

Without loss of generality, we assume that every correct process is always reachable from itself and that all processes with which $p$ has never attempted to communicate are reachable from $p$:

$$p \sim_t q \quad \text{if} \quad p \in \text{Correct}(C) \land p = q \lor \text{last}(p, q, t, \sigma) = \bot.$$ 

A few observations are in order regarding our definitions. First, just like process crashes, partitions are a characterization of a given execution in a distributed system and not of the operating environment. This is only natural since the internal state of components in a distributed system are not observable directly but can only be inferred indirectly through relevant actions generated during an execution. Thus, whether two processes are reachable or partitioned can be revealed only if they attempt to communicate. Second, the definitions are for a given global history, and thus they can base their classification of reachability among process pairs on the outcome of a future event (their last communication attempt). This is no different than classifying a process $p$ as having crashed by time $t$ based on the total absence of future relevant events executed by $p$ in the global history. In practice, the fact that $q$ is reachable from $p$ can be concluded only $a posteriori$ when there is positive confirmation that $p$’s last message to $q$ has been received. Appendix A describes how failure detectors [19] can be built in partitionable systems that provide meaningful estimates of the underlying partition pattern during an execution.

Unlike process crashes, partitions are not permanent but may form and disappear dynamically. This is reflected by the fact that $U(p, t)$ and $U(p, t + 1)$ may differ arbitrarily. Note that in partitionable systems, the partition pattern subsumes the crash pattern in every global history — a process that has crashed by time $t$ will appear to be partitioned from every other process that attempts to communicate with it at time $t$ or later. We nevertheless choose to model crash and partition patterns separately so that specifications can be made in terms of properties that need to hold for correct processes only.

3 Formal Specification of Partitionable Group Membership

Informally, a group membership service tracks changes in the group’s composition and reports them to processes through events that install views. We denote such view changes as $vchg()$ events in global histories. View changes may be due to requests to join and leave the group voluntarily, or they may be due to partitions and merges. For simplicity, we assume that there is a single group and consider changes to its membership due to partitions and merges only. A view installed at process $p$ represents $p$’s perception of the group’s current composition.

Our specification for partitionable group membership service is given as a set of properties on view compositions and view installations, stated in terms of the partition pattern that occurs during an execution. It is highly desirable that the specification be formulated in terms of the actual execution characterization and not in terms of failure detector suspicions, as is done by Neiger [17] and Friedman and Van Renesse [13]. Otherwise, the correctness of the specification itself would be conditional on the properties of the failure detector producing the suspicions. It is reasonable for the correctness of the partitionable group membership service implementation, but not its specification, to be conditional on the correctness of the failure detector implementation.

We start out by defining some terms and introducing notation. Views are identified through unique identifiers drawn from a potentially-infinite domain. Given a view $v$, we write $\pi$ to denote its composition (a set of process names). The event $vchg(v)$ denotes a view change installing view $v$. The current view of process $p$ at time $t$ is $v$, denoted $\text{view}(p, t) = v$, if $v$ is the last view to have been installed at $p$ before time $t$. Events are said to occur in the view that is current. View $w$ is
called the immediate successor of \( v \), denoted \( v \prec w \), if there exists some process \( p \) that installs \( w \) in view \( v \). The successor relation \( \prec^* \) denotes the transitive closure of \( \prec \).

Installed views represent the perception of the group’s composition that is shared by its members. In other words, there has to be agreement among processes on the composition of a view before it can be installed. In a partitionable system, what distinguishes this goal from the traditional consensus problem [12], is the fact that correct processes are not bound by a unique agreement value. A correct process that happens to be partitioned may proceed to agree on a view different from the initial one. These considerations lead to the first property.

**GM1 (View Agreement)** If a correct process \( p \) installs view \( v \), then for every process \( q \) included in \( v \), either (i) \( q \) also installs \( v \), or (ii) \( p \) eventually installs an immediate successor to \( v \) that excludes \( q \). Formally,

\[
\forall t, \ p \in \text{Correct}(C) \land p \rightarrow t \Rightarrow \text{view}(v) \land q \in v \Rightarrow (q \in \text{view}(v) \lor q \notin v).
\]

The composition of installed views cannot be arbitrary but should reflect reality through the partition pattern that verifies during an execution. Informally, each process should install views that include all processes reachable from it and exclude those that are partitioned from it. Requiring that the current view of a process perpetually reflect the actual partition pattern would be impossible to achieve in an asynchronous system. Thus, we state the requirement as two eventual properties.

**GM2 (View Accuracy)** If process \( q \) remains reachable from some correct process \( p \), then eventually the current view of \( p \) will forever include \( q \). Formally,

\[
\exists t_0, \ p \in \text{Correct}(C), \ \forall t \geq t_0 : p \rightarrow t \ q \Rightarrow \exists t_1, \forall t \geq t_1 : q \in \text{view}(p, t).
\]

**GM3 (View Completeness)** If all processes in some set \( \Theta \) remain partitioned from the rest of the system, then eventually the current view of every correct process not in \( \Theta \) will forever exclude all processes in \( \Theta \). Formally,

\[
\exists t_0, \forall q \in \Theta, \forall p \notin \Theta, \forall t \geq t_0 : p \rightarrow t \ q \Rightarrow \exists t_1, \forall t \geq t_1 : \text{view}(p, t) \cap \Theta = \emptyset.
\]

Note that View Accuracy and View Completeness are stated slightly differently. This is because the “reachable” relation between processes is not transitive. While \( q \) being reachable directly from \( p \) is justification for requiring \( p \) to include \( q \) in its view, the converse is not necessarily true. The fact that a process \( p \) cannot communicate directly with another process \( q \) does not imply that \( p \) cannot communicate indirectly with \( q \) through a sequence of pairwise reachable intermediate processes. For this reason, View Completeness has to be stated in terms of complementary sets rather than process pairs. Doing so assures that a process is excluded from a view only if there exists no path, direct or indirect, for reaching it.

Recall that by assumption, every correct process is always reachable from itself. Thus, Property GM2 ensures that eventually, all views installed by a process will include itself. It is, however, desirable that self-inclusion be a perpetual, and not eventual, property of installed views.

**GM4 (View Integrity)** Every view installed by a process includes itself. Formally,

\[
\text{vchg}(v) \in \sigma(p, T) \Rightarrow p \in \text{T}.
\]

The final property for group membership places restrictions on the order in which views are installed. In systems that do not partition, it is reasonable to require that all correct processes
install views in the same order. In a partitionable system, this is not feasible due to the possibility of concurrent partitions. Yet, for a partitionable group membership service to be useful, the set of views must be consistently ordered by those processes that do install them. This can be satisfied by requiring that the successor relation between views does not result in any cycles.

**GM5 (View Order)** The order in which processes install views is such that the successor relation is a partial order. Formally, \( v \prec^* w \Rightarrow w \not\prec^* v \).

Properties GM1–GM5 taken together define a *partitionable group membership service* in asynchronous systems. Note that without Properties GM2 and GM3, the specification can be trivially satisfied by any one of the following useless implementations: (1) At each process \( p \), install only views composed of \( p \) alone, (2) Install views with some fixed, *a priori* agreed upon composition, independent of the actual execution, (3) After a certain time, do nothing (install no views).

The first trivial solution exhibits what has been called *capricious view splitting* [2]. The second trivial solution exhibits *capricious view installation* since processes are added or removed from views without any justification. The third trivial solution lacks liveness. Property GM2 prevents capricious view splitting by requiring that mutually reachable processes appear together in a single view. Capricious view installation is prevented by requiring that the composition of installed views be a function of the actual partition pattern that occurs during an execution, which cannot be predicted ahead of time. At the same time, Properties GM2 and GM3 guarantee liveness since in both cases view installations cannot be delayed indefinitely under stable conditions.

It is important to note that Property GM2 prevents also the following subtle combination of the first and the second trivial solutions: alternating installation of normal views related to the actual partition pattern and singleton views composed of only the installing process. Specifications requiring the *eventual* installation of a new view containing permanently reachable processes is not sufficient to rule out this trivial solution. For this reason, we require that from some time onwards, *all* the views installed by one process will contain the other, as long as they remain reachable.

Note also that Properties GM2 and GM3 exclude trivial solutions even when reachability relations are not stable but are continually changing due to transient failures. In an asynchronous distributed system, no algorithm can distinguish between an execution in which certain processes are eventually partitioned and an execution in which they remain mutually reachable forever. Thus, if the reachability relation between two processes persists for sufficiently long periods, then any implementation satisfying the above specification cannot delay view installation indefinitely but must guarantee that each of the two processes will eventually install a view including the other.

### 4 Solving Partitionable Group Membership

In this section we present an algorithm to implement the specification of Section 3 in partitionable asynchronous systems. Given the inherent complexity of the problem, algorithms to solve it are unavoidably complex. Due to space restrictions, we give abbreviated descriptions without any proofs, which can be found in the full version of the paper [4]. The global structure of the algorithm is shown in Figure 1 and consists of two layers called *Multi-Send Service (MSS)* and *Group Membership Service (GMS)* at each process. FD denotes any failure detector module satisfying the abstract properties for class \( \mathcal{P} \) as defined in Appendix A. The failure detector is used in two distinct ways by our algorithm. First, it provides each process with the initial estimate of group membership to be agreed upon. Second, it is used to render the consensus problem instance solvable within each partition despite asynchrony [19].

All interactions with the communication network and the failure detector are limited to the MSS which uses the unreliable, unsequenced datagram transport service of the network through the
primitives \textit{send()} and \textit{recv()}. Each MSS can also read the suspect list of the corresponding failure detector module. MSS implements primitives \textit{msend()}, \textit{mrecv()} and \textit{msuspect()} as described below, which are in turn used by the GMS. Recall that we consider group membership changes due to partitions and merges only. Thus, the GMS implementation we give includes only the view change notification event \textit{vchg()} but not the primitives \textit{join()} and \textit{leave()} for voluntarily joining and leaving the group. In order to distinguish between the various layers in our discussion, we say that a process \textit{m-sends} and \textit{m-receives} messages when it communicates through the MSS primitives \textit{msend()} and \textit{mrecv()}, respectively. We reserve \textit{send} and \textit{receive} to denote communication directly through the network services without going through the MSS. Similarly, we say that a process \textit{m-suspects} those processes that are notified through a \textit{msuspect()} event and \textit{suspect} is reserved for describing the failure detector itself.

### 4.1 The Multi-Send Service

Implementing a group membership service directly on top of a point-to-point unreliable, unsequenced datagram transport service provided by the network would be difficult. The difficulty is aggravated by the lack of transitivity of the reachability relation as provided by the failure detector. The task of MSS is to hide this complexity by transforming the unreliable, point-to-point network communication primitives to their best-effort, one-to-many counterparts. MSS also “filters” the raw failure detector suspect list by eliminating from it those processes that can be reached indirectly. In other words, the notion of reachability above the MSS corresponds to the transitive closure of reachability at the failure detector layer. What distinguishes MSS from a typical network routing or reliable multicast service is the integration of the delivery semantics with the reachability information. In that sense, MSS is much closer to the \textit{dynamic routing layer} of Phoenix [14] and the MUPS layer of Horus [20].

Here, we describe informally the properties that MSS must satisfy. Formal descriptions along with an algorithm to achieve them can be found in Appendix B. Consider a message $\mu$ m-sent by correct process $p$ to a destination set $G$ through the primitive \textit{msend($\mu$, $G$)}. MSS guarantees that, for each $q$ in $G$, either $\mu$ is m-received by $q$ through a \textit{mrecv($\mu$, $p$)} event, or $p$ m-suspects $q$ through a \textit{msuspect($P$)} event such that $P$ includes $q$. Obviously, this property could be satisfied trivially by every process m-suspecting every other process. To avoid this, MSS has to satisfy non-triviality properties: if a process $q$ is continuously reachable from a process $p$, then $p$ will eventually stop m-suspecting $q$. Furthermore, processes belonging to a given permanent partition must eventually m-suspect processes not included in their partition. MSS guarantees that the reachability relation defined by the \textit{msuspect()} events is eventually symmetric: if a correct process $p$ stops m-suspecting another correct process $q$, then eventually $q$ will stop m-suspecting $p$. Finally, MSS
procedure GroupMembershipService
1  view_id ← \{p, 0\}; view_comp ← \{p\}; reachable ← \{p\}
2  generate vchg((view_id, view_comp))
3  foreach q ∈ \Pi do
4    ctbl[q].seq ← 0
5    version[q] ← 0
6  od
7  version[p] ← 1; seq ← 0
8  while true do
9    wait-for event % Remain idle until some event occurs
10   case event of
11     msuspect(P):
12       msend(\{SYMMETRY, version, reachable\}, (\Pi - P) - reachable)
13       reachable ← \Pi - P
14       AgreementPhase
15     mrecv(\{SYNCHRONIZE, V\}, q):
16       if (version[q] < V[q]) then
17         version[q] ← V[q]
18       if (q ∈ reachable) then
19         AgreementPhase
20       fi
21     esac
22   od
23  procedure AgreementPhase
24  repeat
25    version[p] ← version[p] + 1
26    view_est ← reachable
27    SynchronizationPhase
28    EstimateExchangePhase
29  until (stableview)
30
Figure 2. The main algorithm and agreement phase for process p.

guarantees integrity conditions such that each process m-receives a message at most once and only if some process actually m-sent it earlier; and a process never m-suspects itself. As shown in Appendix B, these requirements can be satisfied by integrating a routing algorithm with a failure detector of class ∞.P.

4.2 The Group Membership Service

In Figures 2–4 we illustrate various components of the algorithm for GMS. Since our aim is to prove that the specification of Section 3 is implementable, the algorithm has been optimized for simplicity rather than efficiency. All messages exchanged are tagged with a type field written in SMALLCAPS. The GMS algorithms are driven by mrecv() and msuspect() events passed up to the algorithm by MSS. The wait-for construct is used to block the algorithm until one of these events occurs.

As can be seen in Figure 2, the algorithm alternates between an idle phase and an agreement phase. A process remains idle until either it is informed by the MSS (through a msuspect() event) that there is a change in the list of m-suspected processes, or it m-receives a message from another process that has observed such a change. Either of these events causes the process to enter the agreement phase whose goal is to establish agreement among the non m-suspected processes towards an identifier and a composition for the next view.

Each process enters the agreement phase with its own estimate for the composition of the next view. During this phase, a process can modify its estimate to reflect changes in the reachability
approximations that are being reported to it by MSS. Each invocation of the agreement phase terminates, even in situations where the reachability relations are highly unstable, with the installation of a new view. This is guaranteed by the fact that successive estimates of each process for the composition of the next view are monotone decreasing sets. This is achieved through two actions. First, new m-suspect lists reported by MSS never cause processes to be added to the initial estimate. Second, processes exchange their estimates with each other and remove those processes that have been removed by others. In this manner, each process continues to reduce its estimate until it coincides exactly with those processes that agree on the composition of the next view. Note that this approach is not in conflict with the View Accuracy requirement of the specification: non-triviality properties of MSS and the construction of the agreement algorithm guarantee that there is a time after which a process can neither m-suspect a permanently reachable process, nor can it m-receive a message inviting it to remove a permanently reachable process. If the new view that is installed coincides with the current set of processes believed to be reachable (as reported by MSS), then the process returns to idle phase (in Figure 2, this condition is encoded as \texttt{stableview}). Otherwise a new agreement is initiated immediately.

When a process enters the agreement phase, it initializes its estimate for the composition of the next view to coincide with those processes that are not included in the most recent m-suspect list supplied by the MSS through its last \texttt{msuspect()} event. The rest of the agreement algorithm is implemented as two phases called \textit{synchronization} and \textit{estimate exchange} as shown in Figures 3 and 4, respectively. The synchronization phase consists of each process m-sending a message containing a sequence number to those processes belonging to its estimate and waiting for a response. In this manner, each process in the estimate of the first process to enter the agreement phase is forced to enter agreement as well and learns the sequence number that will be used to distinguish this agreement phase from others. Estimate exchange phase is entered when a process has m-received an answer to its synchronize request from every process in its estimate.

During this phase, a process may modify its estimate for the composition of the next view, and whenever it does so, it m-sends a message containing the new estimate to each process belonging to the estimate itself. When a process m-receives such a message, it removes from its own

---

**Figure 3.** Synchronization phase of the agreement algorithm for process \( p \).
procedure EstimateExchangePhase
  SendEstimate(∅)
  repeat
    wait-for event
    case event of
      msuspect(π)
        msend((SYMMETRY, version, reachable), (π − P) − reachable)
        reachable ← π − P
      if (view_est ∩ P = ∅) then
        SendEstimate(view_est ∩ P)
      nrecv((SYMMETRY, V, P), q):
        if (version[p] = V[q]) and (q ∈ view_est) then
          SendEstimate(view_est ∩ P)
      nrecv((SYNCHRONIZE, V), q):
        if (version[q] < V[q]) then
          version[q] ← V[q]
      nrecv((EXCLUDE, V, P, R), q):
        if (q ∈ view_est) and (∀r ∈ view_est − P : version[r] = V[r]) then
          if (view_est ∩ P = ∅) then
            SendEstimate(view_est ∩ P)
        elseif (R) then
          msend((EXCLUDE, version, π − view_est, false), view_est − {p})
      nrecv((PROPOSE, S), q):
        if (ctbl[q], seq < S.seq) and (q ∈ view_est) then
          ctbl[q] ← S
        if (∀q, r ∈ ctbl[p], view_est : (ctbl[p].view_est = ctbl[q].view_est) and
            (ctbl[p].version[r] = ctbl[q].version[r])) then
          msend((VIEW, Unique ID(p, seq), ctbl[p].view_est)
      fi
    esac
  until (event = nrecv((VIEW, w, C), q) and (C[p], view_id = view_id) and (q ∈ view_est))
  msend((VIEW, w, C), C[p].view_est)
  S ← {r ∣ ∃s ∈ C[r].view_est ∩ C[r].view_est : C[r].view_id = C[s].view_id} ∩ C[p].view_est
  if P ∈ S then
    view_comp ← S
    view_id ← w
  else
    view_comp ← {r ∣ r ∈ C[p].view_est ∧ C[r].view_id = view_id}
    view_id ← Unique ID(w, view_id)
  fi
  generate vchg((view_id, view_comp))
  stableview ← (view_comp = reachable) and
  (∀q, r ∈ ctbl[p].view_est : ctbl[p].version[r] = ctbl[q].version[r])
procedure SendEstimate(P)
  view_est ← view_est − P
  seq ← seq + 1
  msend((EXCLUDE, version, π − view_est, P = ∅), view_est − {p})
  msend((PROPOSE, (seq, view_id, version, view_est, view_comp)), Coord(view_est));

Figure 4. Estimate exchange phase of the agreement algorithm for process p.
estimate those processes that are excluded from the estimate of the sender. At the same time, each change in the estimate causes a process to m-send an agreement proposal to some process selected among the current estimate (through a deterministic function Coord()) to act as a coordinator. Note that while estimates are evolving, different processes may select different coordinators. Or, the coordinator may crash or be partitioned before the agreement condition has been verified. In all these situations, the current agreement attempt will fail and new estimates will evolve causing a new coordinator to be selected. When the coordinator eventually observes agreement among all of the received proposals, it m-sends a description for the new view consisting of an identifier and the composition that was agreed upon. When a process m-receives this message, it verifies that each of the processes included in the intersection of its previous and the new view have installed the previous view itself. If so, it installs a view composed only by the processes whose insertion does not violate Property GMI. Otherwise, it installs a subview including only the processes belonging to the proposed new view that have actually installed its previous view. Before terminating the procedure, each process verifies if the new current view is stable (i.e. it corresponds to the current set of processes believed to be reachable and no new synchronization messages have been received).

After this introduction, we are ready to analyze in more detail the algorithm. Figure 2 contains the main procedure, composed by the overall initialization of the algorithm and an infinite loop implementing the normal phase and describing the conditions under which a process enters agreement phase. Figure 2 contains the procedure AgreementPhase, that implements the agreement phase. Apart from some initializations, procedure AgreementPhase simply calls procedure SynchronizationPhase (that implements the synchronization phase and is shown in Figure 3) and procedure EstimateExchangePhase (that implements the estimate exchange phase and is shown in Figure 4).

The local state of each process $p$ is defined by the variables $view_{id}$, $view_{comp}$, $reachable$, $version$, $view_{est}$, $received$, $stableview$, $seq$ and $ctbl$. $view_{id}$ and $view_{comp}$ are, respectively, the identifier and the composition of the current view. $reachable$ corresponds to the set of reachable processes (i.e., the processes not m-suspected by the MSS); if this set is different from $view_{comp}$, $p$ begins a new agreement phase. $version$ is an array indexed by $II$; for each $q \in II$, $version[q]$ contains the last version number of $q$ known by $p$. The information contained in $version$ is used to synchronize the processes during the synchronization phase and to avoid processes use obsolete information during an agreement phase. $view_{est}$ is a set of processes containing the proposal for the composition of the next view. $received$ is the set of processes that know the $p$'s version number of the current agreement phase. $stableview$ is a boolean variable indicating whether the last installed view corresponds to the approximation of reachability supplied by the MSS, or the process has to enter agreement phase again. The $seq$ counter is used to obtain FIFO communication with the coordinator, while $ctbl$ (also called coordinator table) contains all the information used by a process when it assumes the role of coordinator; in particular, $ctbl$ is an array of records indexed by $II$, where each record contains the entries $seq$, $view_{id}$, $view_{comp}$, $version$ and $view_{est}$.

All messages that are m-sent contain a tag, indicated as SMALLCAPS, plus other fields relevant to the message type. There are five types of messages. The message $<synchronize, V>$ is used during the synchronization phase. The field $V$ represents the version number array known by the sender at the time of m-sending. When a process enters synchronization phase, it m-sends a synchronize message to each reachable process, that respond with another synchronize message. The version number array is used to discard obsolete messages. The message $<symmetry, V, P>$ is used in both synchronization phase and estimate exchange phase, to handle the case in which the approximation of the reachability obtained by the MSS is temporarily asymmetric. Here, $V$ is again a version number array, while $P$ is the approximation of the set of reachable processes known by $p$. The messages $<exclude, V, P, R>, <propose, S>$ and $<view, w, C>$ are used during the estimate exchange phase. A exclude message is m-sent to the processes belonging to the current estimate whenever this estimate changes. Once again, $V$ is a version number array, while $P$ is the set of processes to be excluded and $R$ is a boolean flag indicating whether the m-receiver
has to respond to the sender with another exclude message. Propose messages are m-sent by
the processes to the coordinator each time the next view estimate changes; the field $S$ contains
the status of the sender, that corresponds to a $ctbl$ entry. Finally, view messages are m-sent by
the coordinator to the processes when the agreement is reached; the field $w$ is the identifier of the
new view, while $C$ is the coordinator table at the time of m-sending, containing the composition
of the view and other information.

5 Conclusions

We have given a formal specification for a partitionable group membership service that ensures
liveness, excludes trivial solutions and is implementable. A service satisfying our specification,
when augmented with view synchrony, forms the foundation for supporting partition-aware ap-
lications [5]. Such applications are characterized by their ability to continue operating even
in the presence of partitions as long as they can reconfigure themselves consistently. A group
membership service provides the necessary properties so that this reconfiguration is possible
and applications can dynamically establish the quality-of-service they can offer each time the
network partitions or merges. The primary partition version of group membership is not suit-
able for supporting partition-aware applications since progress is limited to at most one network
partition.

Our implementation for a partitionable group membership service is based on a failure detector
that has been suitably extended for this environment. The correctness of the implementation is
based solely on the abstract properties of the failure detector and not on the operating character-
istics of the system. As such, the implementation is guaranteed to be correct for any system that
admits the construction of such a failure detector. Any practical failure detector implementation
presents a trade-off between accuracy and responsiveness to failures. By increasing time-out val-
ues after each premature time-out, accuracy is improved but responsiveness suffers. In practice,
to guarantee reasonable responsiveness, time-outs will have finite bounds, perhaps established
dynamically on a per channel or per application basis. Doing so will guarantee that new views
will be installed within an acceptable time after failures at the risk of excluding from the view
some processes that are actually reachable. However, such processes are either very slow them-
selves or have very slow communication links, and thus, it is perfectly reasonable to exclude
them from the view until their delays return to normal levels.

References

[1] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The totem single-
ring ordering and membership protocol. ACM Transactions on Computer Systems, 13(4):311–
membership services. Technical Report TR95-1534, Department of Computer Science, Cor-
nell University, August 1995.
tructure for constructing reliable applications in large-scale distributed systems. In Pro-
ceedings of the 28th Hawaii International Conference on System Sciences, pages 612–621, Maui,
and algorithms. Technical Report UBLCS-97-1, Department of Computer Science, University
network applications. Technical report, Department of Computer Science, University of


A Failure Detectors for Partitionable Systems

In this Appendix, we formalize the stability conditions that are necessary for solving our specification of partitionable group membership as abstract properties of failure detectors that are suitably extended to partitionable systems. Similar failure detector definitions extended for partitionable systems have appeared in other contexts [15, 7]. The failure detector abstraction originally proposed by Chandra and Toueg [19] is for systems with perfectly reliable communication. In partitionable systems, specification of failure detector properties has to be based on reachability between pairs of processes rather than individual processes being correct or crashed. For example, it will be acceptable (and desirable) for the failure detector of $p$ to suspect $q$ that happens to be correct but partitioned from $p$.

Informally, a failure detector is a distributed program that tries to estimate the partition pattern $U$ that occurs in an execution. Each process has access to a local module of the failure detector that monitors a subset of the processes and outputs those that it currently suspects as being partitioned from itself. A failure detector history $H$ is a function from $\mathbb{N} \times T$ to $2^\Pi$ that describes the outputs of the local modules at each process. If $q \in H(p,t)$, we say that $p$ suspects $q$ at time $t$ in $H$. Formally, a failure detector $\mathcal{D}$ is a function that associates with each pair $(C,U)$ a set $\mathcal{D}(C,U)$ denoting failure detector histories that could occur in executions with crash pattern $C$ and partition pattern $U$.

In asynchronous systems, failure detectors are inherently unreliable in that the information they provide may be incorrect. Despite this limitation, failure detectors that satisfy certain completeness and accuracy properties have proven to be useful abstractions for solving practical problems in such systems [19]. Informally, completeness and accuracy state, respectively, the conditions under which a process should and should not be suspected for $H(p,t)$ to be a meaningful estimate of $U(p,t)$. We consider the following adaptations of completeness and accuracy to partitionable systems, maintaining the same names used by Chandra and Toueg for compatibility reasons [19]:

**FD1 (Strong Completeness)** If some process $q$ remains partitioned from correct process $p$, then eventually $p$ will always suspect $q$. Formally, given a crash pattern $C$ and a partition pattern $U$, a failure detector $\mathcal{D}$ satisfies Strong Completeness if all failure detector histories $H \in \mathcal{D}(C,U)$ are such that:

$$p \in \text{Correct}(C) : (\exists t_0, \forall t \geq t_0 : p \not\sim H q) \Rightarrow \exists t_1, \forall t \geq t_1 : q \in H(p,t)).$$

**FD2 (Eventual Strong Accuracy)** If some process $q$ remains reachable from correct process $p$, then eventually $p$ will no longer suspect $q$. Formally, a crash pattern $C$ and an unreachability pattern $U$, a failure detector $\mathcal{D}$ satisfies Eventual Strong Accuracy if all failure detector histories $H \in \mathcal{D}(C,U)$ are such that:

$$p \in \text{Correct}(C) : (\exists t_0, \forall t \geq t_0 : p \sim H q) \Rightarrow \exists t_1, \forall t \geq t_1 : q \not\in H(p,t)).$$

Borrowing from Chandra and Toueg [19], the class of failure detectors satisfying Strong Completeness and Eventual Strong Accuracy are called eventually perfect, and denoted $\mathcal{P}$. In addition to the properties stated above, we can also formulate their weak and perpetual counterparts, thus generating a hierarchy of failure detector classes similar to those of Chandra and Toueg [19]. Informally, weak completeness and accuracy require the corresponding property to hold only for some pair of processes (rather than all pairs), while their perpetual versions require the corresponding property to hold from the very beginning (rather than eventually).

While a detailed discussion of failure detector classes for partitionable systems and reductions between them is beyond the scope of this paper, we make a few brief observations. In asynchronous systems that do not partition, failure detector classes with the weak version of Completeness hap-
pen to be equivalent to those with the strong version. In such systems, it suffices for one correct process to suspect a crashed process since it can (reliably) communicate this information to all other correct processes. In partitionable systems, this is not possible and failure detector classes with weak completeness are strictly weaker than those with strong completeness.

In principle, it is impossible to implement a failure detector \( D \in \Diamond P \) in partitionable asynchronous systems, just as it is impossible to implement a failure detector belonging to any of the classes \( \diamond P, \diamond Q, \diamond S \) and \( \diamond W \) in asynchronous systems with perfectly reliable communication [19]. In practice, however, asynchronous systems are expected to exhibit reasonable behavior and failure detectors for \( \Diamond P \) can indeed be implemented. For example, consider the following algorithm, which is similar to that of Chandra and Toueg [19], but is based on round-trip rather than one-way message time-outs. Each process \( p \) periodically sends a \( p\text{-ping} \) message to every other process in \( P \). When a process \( q \) receives a \( p\text{-ping} \) message, it sends back to \( p \) an \( q\text{-ack} \) message. If process \( p \) does not receive a \( q\text{-ack} \) message within \( \Delta_p(q) \) local time units, \( p \) adds \( q \) to its list of suspects. If \( p \) receives a \( q\text{-ack} \) message from some process \( q \) that it already suspects, \( p \) removes \( q \) from the suspect list and increments its time-out period \( \Delta_p(q) \) for the channel \( (p, q) \).

Note that since processes send \( \text{ack} \) messages only in response to \( \text{ping} \) messages, a process \( p \) will continually time-out on every other process \( q \) that is partitioned from it. Thus, the above algorithm trivially satisfies the Strong Completeness property of \( \Diamond P \) in partitionable asynchronous systems. On the other hand, in an asynchronous system, it is possible for some process \( p \) to observe an unbounded number of premature time-outs for some other process \( q \) even though \( q \) remains reachable from \( p \). In this case, \( p \) would repeatedly add and remove \( q \) from its list of suspects, thus violating the Eventual Strong Accuracy property of \( \Diamond P \). In many practical systems, increasing the time-out period for each communication channel after each mistake will ensure that eventually there are no premature time-outs on any of the communication channels, thus ensuring Eventual Strong Accuracy. Even if time-outs are not increased without bound, resulting in possible incorrect suspicions, our use of failure detectors is such that view installations always terminate, perhaps with a smaller composition. In other words, processes that are either very slow or have very slow communication links will temporarily leave the group and will be merged back in when their delays become smaller.

The only other scenario in which the algorithm fails to achieve Eventual Strong Accuracy occurs when some process \( q \) continues to receive \( p\text{-ping} \) messages but its \( q\text{-ack} \) messages sent to \( p \) are systematically lost. It is reasonable to expect that in practical asynchronous distributed systems this scenario cannot continue indefinitely. Typically, communication channels are bidirectional and rely on the same physical and logical resources in both directions. As a result, the ability or inability to communicate in one direction usually implies that a similar property will eventually hold also in the other direction. Furthermore, typical protocols for implementing the \( \text{send}() \) and \( \text{recv()} \) primitives are based on low-level end-to-end acknowledgements, and thus, require symmetry at the transport layer even for one-way communication. Thus, we assume that the system satisfies Eventual Symmetry as defined below which is sufficient to guarantee Eventual Strong Accuracy:

**Eventual Symmetry** If process \( q \) becomes and remains reachable (partitioned) from \( p \), then eventually \( p \) will become and remain reachable (partitioned) from \( q \) as well. Formally,

\[
\exists t_0, t \geq t_0 : p \sim_{t_0} q \Rightarrow \exists t_1, t \geq t_1 : q \sim_{t_1} p,
\]

\[
\exists t_0, t \geq t_0 : p \not\sim_{t_0} q \Rightarrow \exists t_1, t \geq t_1 : q \not\sim_{t_1} p.
\]

---

6. These are the \( P \equiv Q, S \equiv W, \Diamond P \equiv \Diamond Q \) and \( \Diamond S \equiv \Diamond W \) results of Chandra and Toueg [19].
B Specification and Implementation of the Multi-Send Service

In this Appendix we give a formal specification for MSS and give an algorithm that satisfies it. Recall that MSS services requests made by GMS for m-sending message \( m \) to destination set \( G \) through the primitive \( msend(m, G) \). The messages are globally unique, i.e. each message is m-sent at most once. It exports events \( mrecv(m, p) \) and \( msuspect(P) \) to GMS for m-receiving a message \( m \) from process \( p \) and for m-suspecting processes in \( P \), respectively. Note that m-suspect events are produced spontaneously by MSS and are not solicited explicitly by GMS.

The sequence of reachable sets produced by MSS at each process can be seen as defining a function \( R \) as follows: \( q \in R(p, t) \) if the last \( msuspect(P) \) event generated at process \( p \) by time \( t \) is such that \( q \not\in P \). Properties of MSS that are needed by the group membership algorithm are established by the following definition.

**Definition B.1 (MSS Properties)** The Multi-Send Service (MSS) satisfies the following properties: (a) each process \( m \)-receives a message at most once and only if some process actually \( m \)-sent it earlier; (b) a message that is \( m \)-sent is \( m \)-received by all processes in the destination set that are not \( m \)-suspected; (c) each process \( m \)-receives a message at most once and only if some process actually \( m \)-sent it earlier; (d) a partition eventually \( m \)-suspect processes not included in that partition; (f) if process \( q \) is continuously reachable from \( p \), then \( p \) will eventually stop \( m \)-suspecting \( q \). Formally,

\[
\begin{align*}
(a) & \quad mrecv(m, q) = \sigma(p, t) \Rightarrow mrecv(m, q) \notin \sigma(p, T - \{t\}) \land (\exists G : msend(m, G \cup \{p\}) \in \sigma(q, [0, t])) \\
(b) & \quad msend(m, G \cup \{q\}) = \sigma(p, t) \Rightarrow \exists q' > t : mrecv(m, p) = \sigma(q, t') \land q \notin R(p, t') \lor p \in C(t') \\
(c) & \quad p \in R(p, t) \\
(d) & \quad p, q \in Correct(C), \exists t, \forall t : t_0 : q \in R(p, t) \Rightarrow \exists t, \forall t : t_1 : p \in R(q, t) \\
(e) & \quad \exists \theta, \forall \theta \in \Theta, \forall q \in \Theta, \forall t : t_0 : p \not\in \Theta \Rightarrow \exists t, \forall \theta \in \Theta, \forall t : t_1 : R(r, t) \cap \Theta = \emptyset. \\
(f) & \quad \exists \theta, \forall t : t_0 : p \not\in \Theta \Rightarrow \exists t, \forall t : t_1, \forall q : p \in R(r, t) \Rightarrow q \in R(r, t)
\end{align*}
\]

In Figure 5 we illustrate an algorithm for implementing MSS. Recall that our goal is simply to prove the implementability of the specification and not be concerned about efficiency. Thus, the algorithm uses a simple flooding strategy based on the forwarding of every received message on each output channel. At each process \( p \), MSS maintains a local state defined by the variables reachable, \( fd, ack \) and \( msg \). reachable is the set of processes that are believed to be reachable through direct or indirect paths. This set is constructed from the outputs of the failure detector modules, including remote ones, as they are learned through incoming messages. \( fd, ack \) and \( msg \) are three vectors indexed by \( II \); for every process \( q \in II \), \( fd[q] \) is a set of processes, \( ack[q] \) is a set of messages, while \( msg[q] \) is a vector (indexed by \( II \)) of sets of messages. The variables \( fd, ack \) and \( msg \) can be partitioned in two sections: the local section refers to process \( p \) (\( fd[p], ack[p] \) and \( msg[p] \)); the remote section refers to all other processes (\( fd[q], ack[q] \) and \( msg[q] \), for each \( q \in II - \{p\} \)). Process \( p \) modifies its local section in response to local events (for example, an \( m \)-send request from GMS, the receipt of a message from the network, or a change in the output of the local failure detector module). In particular, \( fd[p] \) records \( p \)'s last reading of the failure detector; \( ack[p] \) is the set of messages that can be acknowledged with the next \( msuspect \) because they have already been m-received by \( p \); \( msg[p][q] \) is the set of messages m-sent by \( p \) to \( q \) for which \( p \) has not yet m-received an acknowledgment. For each process \( q \in II \) different from \( p \), \( fd[q], ack[q] \) and \( msg[q] \) contain \( p \)'s local perception of the corresponding variables of \( q \).

The algorithm is driven by \( recv() \) events from below (network), \( msend() \) events from above (GMS) and local \( tick \) events that are produced periodically. It exports \( mrecv \) and \( msuspect \) events to GMS. At each \( tick \) event, process \( p \) reads the output of the local failure detector module as \( D_p \) and sends a message containing the value of \( msg, ack \) and \( fd \) to all other processes.

When \( p \) receives a message with contents \( \langle M, A, F \rangle \), it verifies if the data regarding \( q \) contained in the message are more recent than the ones saved in the local variables \( msg[q], ack[q] \) and \( fd[q] \).
procedure Multi-Send Service
  foreach q ∈ I do
    foreach r ∈ I do
      msg[q][r] ← \emptyset
      ack[q] ← \emptyset
      fd[q] ← I \{q\}
    od
    reachable ← \{p\}
  od
  reachable ← \{p\}
while true do
  wait-for event
  case event of
    tick:
      fd[p] ← D_p
      foreach q ∈ I \{p\} do
        send((msg, ack, fd), q)
      msend(m, G):
        foreach q ∈ (G ∩ reachable) \{p\} do
          msg[p][q] ← msg[p][q] \cup \{msend, m\}
        if \{p ∈ G\} then
          generate mrecv(m, p)
      recv((M, A, F)):
        foreach q ∈ I \{p\} : UpToDate(M[q], A[q], F[q]) do
        foreach (msend, m) ∈ (M[q][p] \{ack[p]\}) do
          generate mrecv(m, q)
        if \{(suspect, - , G) ∈ (M[q][p] \{ack[p]\}) and (q ∈ reachable)\} then
          generate msuspect(I \{reachable \{G\}\})
          generate msuspect(I \{reachable\})
      fi
    ack[p] ← ack[p] \cup M[q][p]
    msg[p][q] ← msg[p][q] \cup A[q]
  od
  P ← \{p\}; P' = \emptyset
  while (P ≠ P') do
    P' ← P
    P ← \bigcup_{q \in P'} (I \{fd[q]\})
  od
  if \{P ≠ reachable\} then
    foreach q ∈ reachable \{P\} do
      msg[p][q] = \emptyset do
        msg[p][q] ← \{(suspect, Time(), P)\}
    reachable ← P; generate msuspect(I \{P\});
  fi
od

Figure 5. Algorithm for implementing MSS using failure detector D ∈ \mathcal{D}.
To complete the algorithm explanation, a request to m-send message \( m \) to destination set \( G \) is handled in lines 14-16. For each \( q \) in the destination set \( G \), the pair \((\text{msend}, m)\) is inserted into \( \text{msg}[p][q] \) so that it will be sent to all processes at the next \( \text{tick} \) event. If \( p \) belongs to the destination set, then \( m \) is locally m-received. The \text{msend} tag is used to distinguish between messages sent on behalf of the upper layer and those sent internally which have \text{suspect} tags.

We now prove that the algorithm of Figure 5 is correct.

**Lemma B.1 (MSS Properties)** The algorithm of Figure 5 satisfies MSS as specified in Definition B.1.

**Proof.**

(a) We must show that a process generates a \text{mrecv} event for a message \( m \) at most once and only if some process \( q \) actually m-sent it earlier to a set of processes containing \( p \). Before generating the \text{mrecv} event for \( m \), \( p \) checks that \( m \) does not belong to \( \text{ack}_q[p] \); since after the first \text{mrecv} event \( m \) is inserted in \( \text{ack}_q[p] \), \( p \) generates a \text{mrecv} event for \( m \) at most once. As regards the second part, \( p \) generates an event \text{mrecv}(m, q) \) at time \( t_0 \) only after the receipt of a message \( \langle M, A, F \rangle \) such that \( m \in \text{msg}[q][p] \). From the algorithm structure, there is a time \( t_1 < t_0 \) at which \( \text{msg}[q][p] \) contained \( m \); this is possible if and only if \( q \) has executed an event \text{msend}(m, G) \) at time \( t_2 < t_1 \) such that \( p \in G \).

(b) Let \( p \) be a correct process that executes a \text{msend} event for a message \( m \) at time \( t_0 \) to a set of processes containing \( q \); we must show that either (i) \( q \) will eventually generate an \text{mrecv} event for \( m \), or (ii) \( p \) will eventually generate an \text{suspect} event for \( q \). By contradiction, suppose \( q \) never generates a \text{mrecv} event for \( m \) and \( p \) never generates a \text{suspect} event for \( q \) after \( t_0 \). This implies that \( m \) will never be removed from \( \text{msg}[q][p] \) after \( t_0 \). Let \( \Theta \) denote the set of processes that never receive a message \( \langle M, A, F \rangle \) such that \( m \in \text{msg}[q][p] \). From the algorithm, it follows that there is a time \( t_1 > t_0 \) after which for all messages \( \langle M, A, F \rangle \) sent by processes not in \( \Theta \), \( m \in \text{msg}[q][p] \) holds. By the definition of reachability, after \( t_1 \), the process in \( \Theta \) are permanently partitioned from process not in \( \Theta \); since \( q \in \Theta \) and \( p \notin \Theta \), by (e) it follows that there is a time after which every event \text{msuspect}(P) \) generated by \( p \) is such that \( q \in P \).

(c) We must show that no process generates \text{msuspect} events for itself. There are two cases in which a process \( p \) generates a \text{suspect} event: when a change occurs in the reachable set, in which case \( p \) generates a \text{suspect} event for the set \( \Pi - \text{reachable} \), which cannot contain \( p \), or after receiving a message \( \langle M, A, F \rangle \) where \((\text{suspect}, \cdot, G) \in \text{msg}[q][p] \) for some process \( q \). Since \( q \) has inserted the \text{suspect} tuple in \text{msg}[q][p] immediately after having excluded \( p \) from \text{reachable} set and \( G \) is equal to \text{reachable}, \( G \) does not contains \( p \). So, \( p \) never generates a \text{suspect} event for itself.

(d) Let \( p \) be a process that never generates a \text{suspect} event for \( q \) after time \( t_0 \); we must show that there is a time after which \( q \) never generates a \text{suspect} event for \( p \). By contradiction, suppose this is false. By the algorithm, there is a time \( t_1 > t_0 \) at which \( \text{msg}[q][p] \) contains a \((\text{suspect}, N, \cdot)\).
Lemma C.1 \textit{The number of propose messages m-sent by a process} \( p \) \textit{during a view} \( v \) \textit{is bounded.}

\textbf{Proof.} After having entered the estimate exchange phase during view \( v \), process \( p \) \( m \)-sends a propose message if and only if \( p \) modifies its variable \( \text{view} \_\text{est}_p \). By algorithm construction, no process can be added to \( \text{view} \_\text{est}_p \) during an estimate exchange phase; since its starting cardinality is bounded, changes in \( \text{view} \_\text{est}_p \) can cause a finite number of propose messages during a view.

Lemma C.2 If a correct process \( p \) enters synchronization phase during a view \( v \), then \( p \) \textit{will eventually enter estimate exchange phase during} \( v \).

\textbf{Proof.} By contradiction, suppose \( p \) never enters estimate exchange phase during \( v \); this implies that \( p \) never installs new view after \( v \) and definitely stops modifying \( \text{version}_p[p] \). Let \( n \) denote the final value of \( \text{version}_p[p] \). Let \( q \neq p \) be a process belonging to \( \text{view} \_\text{est}_p \) when \( p \) entered synchronization phase during \( v \). If such process does not exist, \( \text{view} \_\text{est}_p = \text{received}_p = \{ p \} \) and the lemma is proved. Otherwise, we have to show that \( q \) will be either inserted in \( \text{received}_p \), or removed from \( \text{view} \_\text{est}_p \). Suppose \( p \) never \( m \)-suspects \( q \) after installing \( v \); otherwise the proof is trivial. When \( p \) enters agreement phase, it \( m \)-sends to \( q \) a synchronize message containing
n. By Lemma B.1(b), q will eventually m-receive the \textit{synchronize} message at time $t_0$ and will register $n$ in $\text{version}_q[p]$. If $q$ m-suspects $p$ after writing $n$, by Lemma B.1(d) and (b), there is a time at which $p$ m-receives from $q$ a message $\langle \text{symmetry}, V, \neg \rangle$ such that $V[p] = n$ and then $p$ removes $q$ from $\text{view}_\text{est}_q$. Thus, suppose $q$ never m-suspects $p$ after $t_0$. There are three possibilities: if $q$ m-receives the message when $p \in \text{view}_\text{est}_q$ and $\text{view}_\text{est}_q \not\subseteq \text{received}_q$, by Lemma B.1(b) and by algorithm construction $p$ will eventually m-receive a \textit{synchronize} message containing $n$ and so it will insert $q$ in $\text{received}_q$. If $q$ m-receives the message when $p \notin \text{view}_\text{est}_q$ and $\text{view}_\text{est}_q \not\subseteq \text{received}_q$, since $q$ does not m-suspect $p$ after $t_0$, then $q$ has m-sent a message $\langle \text{symmetry}, V, \neg \rangle$ such that $V[p] = n$ before $t_0$, and $p$ will eventually m-receive this message and remove $q$ from $\text{view}_\text{est}_q$. If $q$ m-receives the message when $\text{view}_\text{est}_q \subseteq \text{received}_q$, this means that $q$ is in estimate exchange phase. When $q$ installs a new view, it m-sends a \textit{synchronize} message containing $n$ that $p$ will m-receive. By contradiction, suppose $q$ never installs new view. Then, from the algorithm construction $q$ is blocked in estimate exchange phase due to a process $p_1$ different from $p$ that is still in synchronization phase, waiting for a \textit{synchronize} or a \textit{symmetry} message from a process $q_1$. By iterating this construction, we obtain an infinite chain of different processes $p, p_1, p_2, \ldots$, a contradiction that concludes the proof.

**Lemma C.3** If a correct process $p$ enters agreement phase during a view $v$, then it will eventually install a new view.

**Proof.** By contradiction, suppose $p$ enters agreement phase during $v$, but $v$ is the last view installed by $p$. By Lemma C.2, $p$ will eventually enter estimate exchange phase and m-send at least one \textit{propose} message. By Lemma C.1, the number of \textit{propose} messages m-sent by $p$ during $v$ is bounded. Let $\langle \text{propose}, s_p \rangle$ be the last \textit{propose} message m-sent by $p$; let $P$ denote the set $s_p, \text{view}_\text{est}$ and let $n$ denote the value $s_p, \text{version}_p[p]$. By Lemma B.1(c) and by algorithm construction, $P$ is not empty. By Lemma B.1(e), every process in $P$ is correct. By Lemma C.2, there is a time during $v$ at which $\text{view}_\text{est}_p \subseteq \text{received}_p$; thus, every process $q \in P$ has m-received the \textit{synchronize} message m-sent by $p$ during $v$ at the beginning of the agreement phase and has m-sent a message $\langle \text{symmetry}, V_i \rangle$ to $p$ such that $V_i[p] = n$. We claim that after m-sending the \textit{synchronize} message containing $n$, $q$ does not install new views. By contradiction, suppose this is false; if the new view installed by $q$ contains $p$, $p$ will eventually install it after $v$ (contradicting the initial hypothesis); otherwise, by Lemma B.1(d) $p$ will eventually either m-receive a message $\langle \text{symmetry}, V_i, \neg \rangle$ such that $V_i[p] = n$, or m-suspect $q$. In both cases $p$ modifies $\text{view}_\text{est}_p$ and m-sends a new \textit{propose} message, a contradiction that concludes the claim. Then, there is a time after which every process $q \in P$ knows the last value of $\text{version}_q[p]$ and $\text{version}_q[q]$ is equal to the last version number of $q$. Since every process $q \in P$ is in agreement phase and never installs new views, by Lemma C.1 we can denote with $s_q$ the contents of the last \textit{propose} message m-sent by $q$. By applying Lemma C.2 to every process in $P$, we can state that every process in $P$ knows the last version number of each other process in $P$: $\forall q, r \in P: s_q, \text{version}_q = s_r, \text{version}_r$. Moreover, we claim that all the processes in $P$ agree on the value $P$ for the estimate of the composition of the next view: $\forall q, r \in P: s_q, \text{view}_\text{est} = s_r, \text{view}_\text{est}$. By contradiction, suppose this is false: if there is a process $q$ whose estimate contains a process $r \notin P$, then $q$ will eventually m-receive an \textit{exclude} message from $p_i$, accept it and then remove $r$ from its estimate; but this is a contradiction, because $q$ would m-send a new \textit{propose} message. If, on the contrary, there is a process $q$ whose estimate does not contains a process $r \in P$, then $p$ will eventually m-receive a \textit{exclude} or a \textit{symmetry} message from $q$ and accept it; but this is a contradiction, because $p$ would m-send a new \textit{propose} message. So the proof of the claim is concluded. As the $\text{Coord}$ function is univocal, all $\langle \text{propose}, s_q \rangle$ messages have been m-sent to the same coordinator $c = \text{Coord}(P)$ and registered in coordinator table $\text{ctbl}$. After the m-receipt of all the \textit{propose} messages, $c$ reaches agreement on a new view $w$ and m-sends a \textit{view} message to all processes in $P$. When $p$ m-receives the message, it installs the new view, a contradiction that concludes the proof.

**Lemma C.4** If a correct process $p$ installs view $v$, then for every process $q \in \pi$ either (i) $q$ also install $v$,
PROOF. By contradiction, suppose \( q \) never installs \( v \) and \( p \) never enters agreement phase during \( v \). Thus, \( p \) never \( m \)-suspects \( q \) after the installation of \( v \). By Lemma B.1(e), \( q \) is correct; by Lemma B.1(b), \( q \) will eventually \( m \)-receive the \( \text{view} \) message forwarded by \( p \) before installing \( v \). By algorithm construction, if \( q \) does not discard the \( \text{view} \) message, then \( q \) will install \( v \) (a contradiction). Thus \( q \) discharges the \( \text{view} \) message either because it has installed a new \( \text{view} \), or because \( p \) does not belong to \( \text{view}_{\text{est}}_q \). In both cases, \( q \) must have excluded \( p \) from \( \text{view}_{\text{est}}_q \) after \( m \)-sending the \( \text{propose} \) message with which the agreement on \( v \) was reached; after the exclusion, \( q \) has installed a \( \text{view} \) \( w \) not containing \( p \). By Lemma B.1(d), there is a time after which \( q \) stops \( m \)-suspecting \( p \); so, \( q \) will eventually enter agreement phase and \( m \)-send a \( \text{sync} \) message to \( p \). By Lemma B.1(b), \( p \) will eventually receive the message and enter agreement phase during \( v \), a contradiction that concludes the proof.

**Proposition C.1 (View Agreement)** The GMS algorithm satisfies Property GM1.

PROOF. Let \( v \) be a \( \text{view} \) installed by a correct process \( p \) and let \( q \in \pi \) be a process that never installs \( v \). By Lemmas C.4 and C.3, \( p \) will eventually \( m \)-receive a \( \text{view} \) message during \( v \) and install a new \( \text{view} \) \( w \). By algorithm construction, all the processes in \( \pi \cap \bar{\pi} \) have installed \( v \). Thus, \( q \) cannot belong to \( \pi \).

**Proposition C.2 (View Accuracy)** The GMS algorithm satisfies Property GM2.

PROOF. Let \( p \) be a correct process and let \( q \) be always reachable from \( p \) after time \( t_0 \). By Lemma B.1(f), there is a time \( t_1 \) after which every process \( r \) \( m \)-suspects either both or none of \( p \) and \( q \). Thus, \( p \) cannot \( m \)-suspect \( q \) after \( t_1 \) (otherwise it would \( m \)-suspect itself – impossible by Lemma B.1(c)). Thus, \( \text{reachable}_p \) will always contain \( q \) after \( t_1 \). Consider a process \( p_1 \) for which there is a time \( t_{p_1} \) at which \( q \notin \text{view}_{\text{est}}_{p_1} \) and \( p \in \text{view}_{\text{est}}_{p_1} \). There are three possibilities: \( p_1 \) has removed \( q \) after having \( m \)-suspected it, after having \( m \)-received a message \( \langle \text{symmetry}, -, S \rangle \) such that \( q \in S, p \notin S \), or after having \( m \)-received a message \( \langle \text{exclude}, -, S, - \rangle \) from a process \( p_2 \) where \( q \in S, p \notin S \). Consider the third case: there is a time \( t_{p_2} \leq t_{p_1} \) such that \( q \notin \text{view}_{\text{est}}_{p_2} \) and \( p \in \text{view}_{\text{est}}_{p_2} \). By iterating this construction, we can obtain a chain \( p_1, \ldots, p_n \), such that every process \( p_i \) (where \( 1 \leq i < n \)) has \( m \)-received a message \( \langle \text{exclude}, -, S, - \rangle \) from \( p_{i+1} \) where \( q \in S, p \notin S \), and \( p_n \) either has \( m \)-suspected \( q \), but not \( p \), or has \( m \)-received a message \( \langle \text{symmetry}, -, S \rangle \) where \( q \in S, p \notin S \). Note that a process cannot appear twice in the chain \( p_1, \ldots, p_n \), because \( q \) can be removed only once during a \( \text{view} \) and none of the process can have changed \( \text{view} \), due to the fact that before accepting a message \( \langle \text{exclude}, V, -, - \rangle \), each process checks the equality between \( V \) and \( \text{version} \). It follows that an \( \text{exclude} \) message excluding \( q \), but not \( p \), can be propagated a finite number of times. Since after \( t_1 \) none of the processes \( m \)-suspects \( q \) but not \( p \), or \( m \)-sends a message \( \langle \text{symmetry}, -, S \rangle \) with \( q \in S, p \notin S \), there is a time \( t_2 > t_1 \) after which \( p \) cannot removes \( q \) from \( \text{view}_{\text{est}}_p \). Suppose there is a time \( t_3 \geq t_2 \) at which \( q \notin \pi \) and \( v = \text{view}(p, t_3) \) (otherwise the proposition is proved). Since \( \text{reachable}_p \) is different from \( \pi \), \( p \) enters agreement phase during \( v \). By Lemma C.3, \( p \) will receive a \( \langle \text{view}, w, C \rangle \) message after \( t_3 \). Since the \( \text{propose} \) message with which the agreement on \( w \) was concluded can have been \( m \)-sent before \( t_2 \), it is possible that \( q \notin C[p] \text{view}_{\text{est}} \). But every message \( \langle \text{propose}, s_p \rangle \) \( m \)-sent by \( p \) after installing \( w \) is such that \( q \in s_p \text{view}_{\text{est}} \). Thus, every \( \langle \text{view}, w, C \rangle \) message received by \( p \) after \( w \) is such that \( q \in C[p] \text{view}_{\text{est}} \). If \( q \) belongs to the \( \text{view} \) installed by \( p \), the lemma is proved; otherwise, \( q \) enters agreement phase and \( \text{view} \) will eventually install a \( \text{view} \) containing \( q \).

**Proposition C.3 (View Completeness)** The GMS algorithm satisfies Property GM3.

PROOF. Let \( \Theta \) be a set of processes such that there is a time after which all processes in \( \Theta \) are partitioned from processes in \( \Pi - \Theta \). Let \( p \) be a correct process in \( \Pi - \Theta \) and let \( q \) be a process in \( \Theta \); we must prove that there is a time after which every \( \text{view} \) installed by \( p \) does not contain \( q \).
By Lemma B.1(e), there is a time \( t_1 \) after which \( p \) permanently m-suspects \( q \); so, \( q \) is permanently excluded from \( \text{reachable}_p \) and \( \text{view}_{\text{est}}_p \) after \( t_1 \). Suppose there is a time \( t_2 \geq t_1 \) at which \( q \in \tau \) and \( v = \text{view}(p, t_2) \) (otherwise the proposition is proved). Since \( \text{reachable}_p \) is different from \( \tau \), \( p \) enters agreement phase during \( v \). By Lemma C.3, \( p \) will install a new view \( w \) after \( t_2 \). Suppose there is a time \( t_2' \) after \( t_1 \) at which \( q \not\in \text{view}_{\text{est}} \), \( q \) enters another agreement phase and installs a new view not containing \( q \).

**Proposition C.4 (View Integrity)** The GMS algorithm satisfies Property GM4.

**Proof.** Let \( p \) be a process. The first view installed by \( p \) contains only \( p \) itself. Before installing any other view \( v \), \( p \) must m-receive a \( \langle \text{view}, - , C \rangle \) message. By Lemma B.1(a), \( p \) belongs to the destination set of the \text{view} message; this set coincides with \( C[p].\text{view}_{\text{est}} \). By algorithm construction, \( \tau \) contains all the processes in \( C[p].\text{view}_{\text{est}} \) that have installed the previous view of \( p \). Thus, \( p \) belongs to \( \tau \).

**Proposition C.5 (Order)** The GMS algorithm satisfies Property GM5.

**Proof.** The proof is by contradiction. Suppose there are two chains of views, \( v \equiv v_1 \prec \ldots \prec v_n \equiv w \) and \( w \equiv v_{n+1} \prec v_{n+2} \prec \ldots \prec v_m \equiv v \). Let \( t_i \) denote the time at which the coordinator \( c_i \) has m-sent the \text{view} message containing \( v_i \) and let \( p_i \) be a process that has installed \( v_{i-1} \) and \( v_i \) in this order. Let \( t'_{i-1} \) denote the time at which \( p_i \) installed \( v_{i-1} \); obviously, \( t_{i-1} < t'_{i-1} \). When \( p_i \) m-receives the \text{view} message that contains \( v_i \) it verifies that the agreement has been reached with information m-sent by \( p_i \) in \( v_{i-1} \) and therefore \( t'_{i-1} < t_i \); thus, \( t_{i-1} < t_i \). By transitivity, we can state that \( t_1 < t_n \) and \( t_n < t_m = t_1 \), a contradiction that concludes the proof.