Group Communication in Partitionable Systems: Specification and Algorithms

Özalp Babaoğlu, Renzo Davoli, and Alberto Montresor

Università di Bologna, Mura Anteo Zamboni 7, I-40127 Bologna (Italy)
{babaoglu,davoli,montresor}@cs.unibo.it,
WWW home page: http://www.cs.unibo.it/projects/relacs.html

Abstract. We give a formal specification and an implementation for a partitionable group communication 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 guarantees liveness and excludes trivial solutions; it constitutes a useful basis for building realistic partition-aware applications; and it is implementable in practical asynchronous distributed systems where certain stability conditions hold.

1 Introduction

Functional requirements, which define how output values are related to input values, are usually sufficient for specifying traditional applications. For modern network applications, however, non-functional requirements can be just as important as their functional counterparts: the services that these applications provide must not only be correct with respect to input-output relations, they must also be delivered with acceptable "quality" levels. Reliability, timeliness and configurability are examples of non-functional requirements that are of particular interest to network applications.

A correct application satisfies its functional requirements in all possible operating environments: it just may take more or less time to do so depending on the characteristics of the environment. On the other hand, there may be operating environments in which it is impossible to achieve non-functional properties beyond certain levels. For this reason, non-functional requirements of network applications define acceptable quality *intervals* rather than exact values. In order to deliver quality levels that are both feasible and acceptable, network applications need to be *environment aware* such that they can dynamically modify their behavior depending on the properties of their operating environment.

By their nature, network applications for mobile computing, data sharing or collaborative work involve cooperation among multiple sites. For these applications, which are characterized by reliability and configurability requirements, possible partitionings of the communication network is an extremely important aspect of the environment. In addition to accidental partitionings caused by failures, mobile computing systems typically support "disconnected operation" which is nothing more than a voluntary partitioning caused by deliberately unplugging units from the network. The nature of a partitioning will determine the quality for the application in terms of which of its services are available where, and at what performance levels. In other words, partitionings may result in service *reduction* or service *degradation* but need not necessarily render application services completely unavailable. Informally, we define the class of *partition-aware* applications as those that are able to make progress in multiple concurrent partitions without blocking.

Service reduction and degradation that are unavoidable during partitionings depend heavily on the application semantics and establishing them for arbitrary applications is beyond the scope of this chapter. For certain application classes with strong consistency requirements, it may be the case that all services have to be suspended completely in all but one partition. This situation corresponds to the so-called *primary-partition* model [32, 22] that has traditionally characterized partitioned operation of network applications. In this chapter we focus on the specification and implementation of system services for supporting partition awareness such that continued operation of network applications is not restricted to a single partition but may span multiple concurrent partitions. Our goal is for the system to provide only the necessary mechanisms without imposing any policies that govern partitioned operation. In this manner, each application itself can decide which of its services will be available in each partition and at what quality levels.

Our methodology for partition-aware application development is based on the *process group* paradigm [22, 8] suitably extended to partitionable systems. In this methodology, processes that cooperate in order to implement a given network application join a named group as members. All events that are relevant for partition awareness (process crashes and recoveries, network partitionings and merges) are unified in a single abstraction: the group's current membership. At each process, a *partitionable group membership service* installs *views* that correspond to the process's local perception of the group's current membership. Partition-aware applications are programmed so as to reconfigure themselves and adjust their behavior based on the composition of installed views. In a partitionable system, a group membership service has to guarantee that processes within the same partition install identical views and that their composition corresponds to the partition itself. Otherwise, inconsistencies may compromise functional correctness or may result in quality levels that are lower than what is feasible.

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 [3]. Formal specification of a partitionable group membership service in an asynchronous system has proven to be elusive and numerous prior attempts have been unsatisfactory [29, 1, 14–17, 5, 33]. Anceaume *et al.* discuss at length the shortcomings of previous attempts [3]. In summary, existing specifications admit solutions that suffer from one or all of the following problems: (i) they are informal or ambiguous [33, 5, 16], (ii) they cease to install new views even in cases where the group membership continues to change [17], (iii) they capriciously split the group into several concurrent views, possibly down to singleton sets [29, 1, 14, 15, 17], (iv) they capriciously install views without any justification from the operating environment [14, 15]. 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.

In this chapter, we give a formal specification for partitionable group membership services that has the following desirable properties: (i) it does not suffer from any of the problems that have been observed for previous solutions, (ii) it is implementable in asynchronous distributed systems that exhibit certain stability conditions which we formally characterize, (iii) it is useful in that it constitutes the basis for system abstractions that can significantly simplify the task of developing realistic partition-aware applications. To "prove" the usefulness of a collection of new system abstractions, one would need to program the same set of applications twice: once using the proposed abstractions and a second time without them, and compare their relative difficulty and complexity. In another paper, we have pursued this exercise by programming a set of practical partition-aware applications on top of a *group communication service* based on our group membership specification extended with a reliable multicast service with *view synchrony* semantics [7]. For this reason, the current paper is limited to the specification of these services and their implementability.

The rest of the chapter is organized as follows. In the next section, we introduce the system model and define basic properties of communication in the presence of partitions. In Sect. 3 we give a formal specification for partitionable group membership services that guarantees liveness and excludes useless solutions. In Sect. 4 we extend the *failure detector* abstraction of Chandra and Toueg [11] to partitionable systems and show how it can be implemented in practical asynchronous systems where certain stability conditions hold. In Sect. 5 we prove that our specification is implementable on top of an unreliable datagram communication service in systems that admit failure detectors. In Section 6 we briefly illustrate how our partitionable group membership service may be extended to a group communication service based on view synchrony. Section 7 relates our specification to numerous other proposals for group communication and Sect. 8 concludes the work.

2 System Model

We adopt notation and terminology similar to that of Chandra and Toueg [11]. The system comprises a set Π of processes that can communicate by exchanging messages through a network. Processes are associated unique names that they maintain throughout their life. The communication network implements channels connecting pairs of processes and the primitives *send*() and *recv*() for sending and receiving messages over them. 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 $\mathcal{T}$. This simplification is not in conflict with the asynchrony assumption since processes are never allowed to access the global clock.

2.1 Global Histories

The execution of a distributed program results in each process performing an event (possibly null), chosen from a set $\mathcal{S}$, at each clock tick. Set $\mathcal{S}$ includes at least the events *send*() and *recv*() corresponding to their respective communication primitives. In Sect. 3 we extend this set with other events related to group membership. The *global history* of an execution is a function σ from $\Pi \times \mathcal{T}$ to $\mathcal{S} \cup \{\epsilon\}$, where ϵ denotes the null event. If process p executes an event $e \in \mathcal{S}$ at time t, then $\sigma(p,t) = e$. Otherwise, $\sigma(p,t) = \epsilon$ indicating that process p performs no event at time t. Given some interval $\mathcal{I}$ of $\mathcal{T}$, we write $e \in \sigma(p,\mathcal{I})$ if p executes event e sometime during interval $\mathcal{I}$ of global history σ (i.e, $\exists t \in \mathcal{I} : \sigma(p,t) = e$).

2.2 Communication Model

In the absence of failures, the network is logically connected and each process can communicate with every other process. A process p sends a message m to a process q by executing $\mathit{send}(m,q)$, and receives a message m that has been sent to it by executing $\mathit{recv}(m)$. Communication is unreliable (as described below) and sequencing among multiple messages sent to the same destination need not be preserved (i.e., channels are not FIFO). Without loss of generality, we assume that (i) all messages sent are globally unique, and (ii) a message is received only if it has been previously sent. Note that this communication model is extremely faithful to practical distributed systems built on top of typical unreliable datagram transport services such as IP and UDP.

2.3 Failure Model

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 an execution is captured through the *crash pattern* function C from $\mathcal{T}$ to 2^{Π} where $C(t)$ denotes the set of processes that have crashed by time t. Since crashed processes do not recover, we have $C(t) \subseteq C(t+1)$. With $\mathit{Correct}(C) = \{p \mid \forall t : p \notin C(t)\}$ we denote those processes that never crash, and thus, are correct in C.

A variety of events, including link crashes, buffer overflows, incorrect or inconsistent routing tables, may disable communication between processes. We refer to them generically as *communication failures*. Unlike process crashes, which are permanent, communication failures may be temporary due to subsequent repairs. The evolution of communication failures and repairs during an execution is captured through the *unreachability pattern* function U from $\Pi \times \mathcal{T}$ to 2^{Π} where $U(p,t)$ denotes the set of processes with which p cannot communicate at time t. If $q \in U(p,t)$, we say that process q is *unreachable* from p at time t, and write $p \not\rightsquigarrow_t q$ as a shorthand; otherwise we say that process q is *reachable* from p at time t, and write $p \rightsquigarrow_t q$. As noted above, communication failures are not necessarily permanent but may appear and disappear dynamically. This is reflected by the fact that the sets $U(p,t)$ and $U(p,t+1)$ may differ arbitrarily.

Note that the unreachability pattern is an abstract characterization of the communication state of a system, just as the crash pattern is an abstract characterization of its computational state. Only an omnipotent external observer can construct the unreachability and crash patterns that occur during an execution and neither can be inferred from within an asynchronous system. Nevertheless, they are useful in stating desired properties for a group membership service. Any implementation of the specified service in an asynchronous system will have to be based on approximations of unreachability and crashes provided by *failure detectors* [11] as we discuss in Sect. 4.

Reachable/unreachable are attributes of individual communication channels (identified as ordered process pairs), just as correct/crashed are attributes of individual processes. In the rest of the chapter, we also refer to communication failure scenarios called *partitionings* that involve multiple sets of processes. A partitioning disables communication among different *partitions*, each containing a set of processes. Processes within a given partition can communicate among themselves, but cannot communicate with processes outside the partition. When communication between several partitions is reestablished, we say that they *merge*.

Process and communication failures that occur during an execution are not totally independent, but must satisfy certain constraints that are captured through the notion of a *failure history*:

Definition 1 (Failure History). *A failure history F is a pair (C,U), where C is a crash pattern and U is an unreachability pattern, such that (i) a process that has crashed by time t is unreachable from every other process at time t, and (ii) a process that has not crashed by time t is reachable from itself at time t. Formally,*[1]

$$\begin{aligned} (i)\ p \in C(t) \ &\Rightarrow\ q \not\rightsquigarrow_t p \\ (ii)\ p \notin C(t) \ &\Rightarrow\ p \rightsquigarrow_t p \ . \end{aligned}$$

By definition, the unreachability pattern subsumes the crash pattern in every failure history. We nevertheless choose to model crash and unreachability pat-

[1] In these formulas and all others that follow, free variables are assumed to be universally quantified over their respective domains (process events, time, messages, views, etc.), which can be inferred from context.

terns separately so that specifications can be made in terms of properties that need to hold for *correct* processes only.

Finally, we need to relate crash and unreachability patterns to the events of the execution itself. In other words, we need to formalize notions such as "crashed processes halt prematurely" and "unreachable processes cannot communicate directly". We do this by requiring that the global and failure histories of the same execution conform to constraints defining a *run*.

Definition 2 (Run). *A run R is a pair (σ, F), where σ is a global history and $F = (C, U)$ is the corresponding failure history, such that (i) a crashed process stops executing events, and (ii) a message that is sent will be received if and only if its destination is reachable from the sender at the time of sending. Formally,*

$$\begin{aligned}&(i)\ p \in C(t) \ \Rightarrow\ \forall t' \geq t : \sigma(p, t') = \epsilon \\ &(ii)\ \sigma(p, t) = \mathit{send}(m, q) \ \Rightarrow\ (\ \mathit{recv}(m) \in \sigma(q, \mathcal{T}) \Leftrightarrow p \rightsquigarrow_t q\ )\ .\end{aligned}$$

Note that by Definition 1(ii), the reachable relation for correct processes is *perpetually* reflexive — a correct process is always reachable from itself. Transitivity of reachability, on the other hand, need not hold in general. We make this choice so as to render our model realistic by admitting scenarios that are common in wide-area networks, including the Internet, where a site B may be reachable from site A, and site C reachable from B, at a time when C is unreachable from A directly. Yet the three sites A, B and C should be considered as belonging to the same partition since they *can* communicate with each other (perhaps indirectly) using communication services more sophisticated than the send/receive primitives offered by the network. As we shall see in Sect. 5.1, such services can indeed be built in our system model so that two processes will be able to communicate with each other whenever it is possible. And our notion of a partition as the set of processes that can mutually communicate will be based on these services.

We do not assume perpetual symmetry for the reachable relation. In other words, at a given time, it is possible that some process p be reachable from process q but not vice versa. This is again motivated by observed behavior in real wide-area networks. Yet, to make the model tractable, we require a form *eventual* symmetry as stated below:

Property 1 (Eventual Symmetry). If, after some initial period, process q becomes and remains reachable (unreachable) from p, then eventually p will become and remain reachable (unreachable) from q as well. Formally,

$$\begin{aligned}\exists t_0, \forall t \geq t_0 : p \rightsquigarrow_t q \ &\Rightarrow\ \exists t_1, \forall t \geq t_1 : q \rightsquigarrow_t p \\ \exists t_0, \forall t \geq t_0 : p \not\rightsquigarrow_t q \ &\Rightarrow\ \exists t_1, \forall t \geq t_1 : q \not\rightsquigarrow_t p\ .\end{aligned}$$

This is a reasonable behavior to expect of practical asynchronous distributed systems. 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 be observed also in the other direction.

To conclude the system model, we impose a fairness condition on the communication network so as to exclude degenerate scenarios where two processes are unable to communicate despite the fact that they become reachable infinitely often. In other words, the communication system cannot behave maliciously such that two processes that are normally reachable become unreachable precisely at those times when they attempt to communicate.

Property 2 (Fair Channels). Let p and q be two processes that are not permanently unreachable from each other. If p sends an unbounded number of messages to q, then q will receive an unbounded number of these messages. Formally,

$$(\forall t, \exists t_1 \geq t : p \rightsquigarrow_{t_1} q) \wedge (\forall t, \exists t_2 \geq t : \sigma(p, t_2) = \mathit{send}(m, q)) \;\Rightarrow\; (\forall t, \exists t_3 \geq t : \sigma(q, t_3) = \mathit{recv}(m') \wedge \mathit{send}(m', q) \in \sigma(p, \mathcal{T})) \;.$$

3 Partitionable Group Membership Service: Specification

Our methodology for partition-aware application development is based on the *process group* paradigm with suitable extensions to partitionable systems. In this methodology, processes cooperate towards a given network application by *joining* a group as members. Later on, a process may decide to terminate its collaboration by explicitly *leaving* the group. In the absence of failures, the *membership* of a group comprises those processes that have joined but have not left the group. In addition to these voluntary events, membership of a group may also change due to involuntary events corresponding to process and communication failures or repairs.

At each process, a *partitionable group membership service* (PGMS) tracks the changes in the group's membership and installs them as *views* through *vchg*() events. Installed views correspond to the process's local perception of the group's current membership. Partition-aware applications are programmed so as to reconfigure themselves and adjust their behavior based on the composition of installed views. In the absence of partitionings, every correct process should install the same view, and this view should include exactly those members that have not crashed. This goal is clearly not feasible in a partitionable system, where processes in different partitions will have different perceptions of the membership for a given group. For these reasons, a partitionable group membership service should guarantee that under certain stability conditions, correct processes within the same partition install identical views and that their composition correspond to the composition of the partition itself.

In the next section, we translate these informal ideas in a formal specification for our partitionable group membership service. The specification is given as a set of properties on view compositions and view installations, stated in terms of the unreachability pattern that occurs during an execution. The specification we give below has benefited from extensive reflection based on actual experience with programming realistic applications and has gone through numerous refinements over the last several years. We believe that it represents a minimal set of properties for a service that is both useful and implementable.

3.1 Formal Specification

For sake of brevity, we assume a single process group and do not consider changes to its membership due to voluntary join and leave events. Thus, the group's membership will vary only due to failures and repairs. We start out by defining some terms and introducing notation. Views are labeled in order to be globally unique. Given a view v, we write $\overline{v}$ to denote its composition as a set of process names. The set of possible events for an execution, $\mathcal{S}$, is augmented to include $\mathit{vchg}(v)$ denoting a view change that installs view v. The *current view* of process p at time t is v, denoted $\mathit{view}(p,t) = v$, 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 *immediate successor of v at p*, denoted $v \prec_p w$, if p installs w in view v. View w is called *immediate successor* of v, denoted $v \prec w$, if there exists some process p such that $v \prec_p w$. The *successor* relation $\prec^*$ denotes the transitive closure of $\prec$. Two views that are not related through $\prec^*$ are called *concurrent.* Given two immediate successor views $v \prec w$, we say that a process *survives* the view change if it belongs to both v and w.

The composition of installed views cannot be arbitrary but should reflect reality through the unreachability pattern that occurs during an execution. In other words, processes should be aware of other processes with which they can and cannot communicate directly in order to adapt their behaviors consistently. Informally, each process should install views that include all processes reachable from it and exclude those that are unreachable from it. Requiring that the current view of a process perpetually reflect the actual unreachability pattern would be impossible to achieve in an asynchronous system. Thus, we state the requirement as two eventual properties that must hold in stable conditions where reachability and unreachability relations are persistent.

GM1 (View Accuracy). *If there is a time after which process q remains reachable from some correct process p, then eventually the current view of p will always include q. Formally,*

$$\exists t_0,\ \forall t \geq t_0 : p \in \mathit{Correct}(C) \wedge p \rightsquigarrow_t q \ \Rightarrow\ \exists t_1, \forall t \geq t_1 : q \in \overline{\mathit{view}(p,t)}\ .$$

GM2 (View Completeness). *If there is a time after which all processes in some partition Θ remain unreachable from the rest of the group, then eventually the current view of every correct process not in Θ will never include any process in Θ. Formally,*

$$\exists t_0, \forall t \geq t_0, \forall q \in \Theta, \forall p \notin \Theta : p \not\rightsquigarrow_t q \ \Rightarrow$$
$$\exists t_1, \forall t \geq t_1, \forall r \in \mathit{Correct}(C) - \Theta : \overline{\mathit{view}(r,t)} \cap \Theta = \emptyset\ .$$

View Accuracy and View Completeness are of fundamental importance for every PGMS. They state that the composition of installed views cannot be arbitrary but must be a function of the actual unreachability pattern occurring during a run. Any specification that lacked a property similar to View Accuracy could be trivially satisfied by installing at every process either an empty view or a singleton view consisting of the process itself. The resulting service would exhibit what has been called *capricious view splitting* [3] and would not be very useful. View Accuracy prevents capricious view splitting by requiring that eventually, all views installed by two permanently-reachable processes contain each other. On the other hand, the absence of View Completeness would admit implementations in which processes always install views containing the entire group, again rendering the service not very useful.

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 of processes rather than process pairs. Doing so assures that a process is excluded from a view only if communication is impossible because there exists no path, directly or indirectly, for reaching it.

View Accuracy and View Completeness state requirements for views installed by individual processes. A group membership service that is to be useful must also place constraints on views installed by different processes. Without such coherency guarantees for views, two processes could behave differently even though they belong to the same partition but have different perceptions of its composition. For example, consider a system with two processes p and q that are permanently reachable from each other. By View Accuracy, after some time t, both p and q will install the same view v containing themselves. Now suppose that at some time after t, a third process r becomes and remains reachable from q alone. Again by View Accuracy, q will eventually install a new view w that includes r in addition to itself and p. Presence of process r is unknown to p since they are not directly reachable. Thus, p continues believing that it shares the same view with q since its current view v continues to include q, when in fact process q has gone on to install view w different from v. The resulting differences in perception of the environment could lead processes p and q to behave differently even though they belong to the same partition. The following property has been formulated to avoid such undesirable scenarios.

GM3 (View Coherency).
(i) If a correct process p installs view v, then either all processes in $\overline{v}$ also install v, or p eventually installs an immediate successor to v. Formally,

$$p \in \mathit{Correct}(C) \wedge \mathit{vchg}(v) \in \sigma(p, \mathcal{T}) \wedge q \in \overline{v} \;\Rightarrow$$
$$(\mathit{vchg}(v) \in \sigma(q, \mathcal{T})) \vee (\exists w : v \prec_p w) \ .$$

(ii) If two processes p and q initially install the same view v and p later on installs an immediate successor to v, then eventually either q also installs an immediate successor to v, or q crashes. Formally,

$$vchg(v) \in \sigma(p, \mathcal{T}) \wedge vchg(v) \in \sigma(q, \mathcal{T}) \wedge v \prec_p w_1 \wedge q \in Correct(C) \;\Rightarrow\; \exists w_2 : v \prec_q w_2 \;.$$

(iii) When process p installs a view w as the immediate successor to view v, all processes that survive from view v to w along with p have previously installed v. Formally,

$$\sigma(p, t_0) = vchg(w) \wedge v \prec_p w \wedge q \in \overline{v} \cap \overline{w} \wedge q \neq p \;\Rightarrow\; vchg(v) \in \sigma(q, [0, t_0[) \;.$$

Returning to the above example, the current view of process p cannot remain v indefinitely as GM3(ii) requires p to eventually install a new view. By assumption, q never installs another view after w. Thus, by GM3(i), the new installed by p must be w as well and include r. As a result, processes p and q that belong to the same partition return to sharing the same view. In fact, we can generalize the above example to argue that View Coherency together with View Accuracy guarantee that every view installed by a correct process is also installed by all other processes that are permanently reachable from it. Note that the composition of the final view installed by p and q includes process r as belonging to their partition. This is reasonable since p and r can communicate (using q as a relay) even though they are not reachable directly.

View Coherency is important even when reachability and unreachability relations are not persistent. In these situations where View Accuracy and View Completeness are not applicable, View Coherency serves to inform a process that it no longer shares the same view with another process. Consider two processes p and q that are initially mutually reachable. Suppose that p has installed a view v containing the two of them by some time t. The current view of process q could be different from v at time t either because it never installs v (e.g., it crashes) or because it installs another view after having installed v (e.g., there is a network partitioning or merge). In both cases, GM3(i) and GM3(ii), respectively, ensure that process p will eventually become aware of this fact because it will install a new view after v.

When a process installs a new view, it cannot be sure which other processes have also installed the same view. This is an inherent limitation due to asynchrony and possibility of failures. GM3(iii) allows a process to reason *a posteriori* about other processes: At the time when process p installs view w as the immediate successor of view v, it can deduced which other processes have also installed view v. And if some process q belonging to view v never installs it, we can be sure that q cannot belong to view w. Note that these conclusions are based entirely on *local* information (successive pairs of installed views) yet they allow a process to reason globally about the actions of other processes.

The next property for group membership places restrictions on the order in which views are installed. In systems where partitionings are impossible, it is reasonable to require that all correct processes install views according to some total 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. In other words, if two views are installed by a process in a given order, the same two views cannot be installed in the opposite order by some other process.

GM4 (View Order). *The order in which processes install views is such that the successor relation is a partial order. Formally,* $v \prec^* w \;\Rightarrow\; w \not\prec^* v$.

When combined with View Accuracy and View Coherency, View Order allows us to conclude that there is a time after which permanently reachable processes not only install the same set of views, they install them in the same order.

The final property of our specification places a simple integrity restriction on the composition of the views installed by a process. By Definition 1(ii), every correct process is always reachable from itself. Thus, Property GM1 ensures that eventually, all views installed by a process will include itself. However, it is desirable that self-inclusion be a perpetual, and not only eventual, property of installed views.

GM5 (View Integrity). *Every view installed by a process includes the process itself. Formally,*

$$vchg(v) \in \sigma(p, \mathcal{T}) \;\Rightarrow\; p \in \overline{v} \ .$$

Properties GM1–GM5 taken together define a *partitionable group membership service* (PGMS).

3.2 Discussion

Recall that Properties GM1 and GM2 are stated in terms of runs where reachability and unreachability relations are persistent. They are, however, sufficient to exclude trivial solutions to PGMS also in runs where reachability and unreachability among processes are continually changing due to transient failures. As an example, consider a system composed of two processes p and q and a run R_0 where they are permanently mutually reachable. By View Accuracy and View Coherency, we know there is a time t_0 by which both p and q will have installed a view composed of themselves alone. Now, consider run R_1 identical to R_0 up to time $t_1 > t_0$ when p and q become unreachable. The behavior of processes p and q under runs R_0 and R_1 must be identical up to time t_1 since they cannot distinguish between the two runs. Thus, if they install views composed of p and q by time t_0 under run R_0, they must install the same views also under run R_1 where reachability relations are not persistent but transient. This example can be generalized to conclude that any implementation satisfying our specification cannot delay arbitrarily installation of a new view including processes that remain reachable for sufficiently long periods. Nor can it delay arbitrarily installation of a new view excluding processes that remain unreachable for sufficiently long periods.

Asynchronous distributed systems present fundamental limitations for the solvability of certain problems in the presence of failures. Consensus [19] and primary-partition group membership [10] are among them. Partitionable group membership service, as we have defined it, happens to be not solvable in an asynchronous system as well. A proof sketch of this impossibility results can be found in the extended version of the paper [6]. The impossibility result for PGMS can be circumvented by requiring certain stability conditions to hold in an asynchronous system. In the next section we formulate these conditions as abstract properties of an unreliable failure detector [11]. Then in Sect. 5 we show how the specified PGMS can be implemented in systems that admit the necessary failure detector.

4 Failure Detectors for Partitionable Systems

In this section, we formalize the stability conditions that are necessary for solving our specification of partitionable group membership in asynchronous systems. We do so indirectly by stating a set of abstract properties that need to hold for failure detectors that have been suitably extended to partitionable systems. Similar failure detector definitions extended for partitionable systems have appeared in other contexts [25, 12]. The failure detector abstraction originally proposed by Chandra and Toueg [11] 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 is unreachable from p.

Informally, a failure detector is a distributed oracle that tries to estimate the unreachability 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 unreachable from itself. A *failure detector history* H is a function from $\Pi \times \mathcal{T}$ to 2^{Π} 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 failure history $F = (C, U)$ a set $\mathcal{D}(F)$ denoting failure detector histories that could occur in executions with failure history F.

In asynchronous systems, failure detectors are inherently unreliable in that the information they provide may be incorrect. Despite this limitation, failure detectors satisfying certain *completeness* and *accuracy* properties have proven to be useful abstractions for solving practical problems in such systems [11]. 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 [11]:

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

$$\exists t_0, \forall t \geq t_0 : p \in \mathit{Correct}(C) \wedge p \not\rightsquigarrow_t q \;\Rightarrow\; \exists t_1, \forall t \geq t_1 : q \in H(p,t) \;.$$

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

$$\exists t_0, \forall t \geq t_0 : p \in \mathit{Correct}(C) \wedge p \rightsquigarrow_t q \;\Rightarrow\; \exists t_1, \forall t \geq t_1 : q \notin H(p,t) \;.$$

Borrowing from Chandra and Toueg [11], failure detectors satisfying Strong Completeness and Eventual Strong Accuracy are called *eventually perfect*, and their class denoted $\diamond\tilde{\mathcal{P}}$. 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 [11]. 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 hierarchy for partitionable systems and reductions between them is beyond the scope of this chapter, we make a few brief observations. In absence of partitionings, failure detector classes with the weak version of Completeness happen to be equivalent to those with the strong version. [2] 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 $\mathcal{D} \in \diamond\tilde{\mathcal{P}}$ in partitionable asynchronous systems, just as it is impossible to implement a failure detector belonging to any of the classes $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ in asynchronous systems with perfectly-reliable communication [11]. In practice, however, asynchronous systems are expected to exhibit reasonable behavior and failure detectors for $\diamond\tilde{\mathcal{P}}$ can indeed be implemented. For example, consider the following algorithm, which is similar to that of Chandra and Toueg [11], but is based on *round-trip* rather than *one-way* message time-outs. Each process p periodically sends a p-ping message to every other process in Π. When a process q receives a p-ping, it sends back to p a q-ack message. If process p does not receive a q-ack within $\Delta_p(q)$ local time units, p adds q to its list of suspects. If p receives a q-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).

[2] These are the $\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$ and $\diamond\mathcal{S} \cong \diamond\mathcal{W}$ results of Chandra and Toueg [11].

Note that since processes send *ack* messages only in response to *ping* messages, a process p will continually time-out on every other process q that is unreachable from it. Thus, the above algorithm trivially satisfies the Strong Completeness property of $\diamond\tilde{\mathcal{P}}$ 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 $\diamond\tilde{\mathcal{P}}$. 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.

The only other scenario in which the algorithm could fail to achieve Eventual Strong Accuracy occurs when process q is reachable from process p and continues to receive p-ping messages but its q-ack messages sent to p are systematically lost. In a system satisfying Eventual Symmetry, this scenario cannot last forever and eventually p will start receiving q-ack messages, causing it to permanently remove q from its suspect list and thus satisfy Eventual Strong Accuracy.

Given that perfectly reliable failure detectors are impossible to implement in asynchronous systems, it is reasonable to ask: what are the consequences of mistakenly suspecting a process that is actually reachable? As we shall see in the next section, our use of failure detectors in solving PGMS is such that incorrect suspicions may cause installation of views smaller than what are actually feasible. In other words, they may compromise View Accuracy but cannot invalidate any of the other properties. As a consequence, processes that are either very slow or have very slow communication links may be temporarily excluded from the current view of other processes to be merged back in when their delays become smaller. This type of "view splitting" is reasonable since including such processes in views would only force the entire computation to slow down to their pace. Obviously, the notion of "slow" is completely application dependent and can only be established on a per-group basis.

5 Partitionable Group Membership Service: Implementation

In this section we present an algorithm that implements the service specified in Sect. 3 in partitionable asynchronous systems augmented with a failure detector of class $\diamond\tilde{\mathcal{P}}$. Our goal is to show the implementability of the proposed specification for PGMS; consequently, the algorithm is designed for simplicity rather than efficiency. The overall structure of our solution is shown in Fig. 1 and consists of two components called the *Multi-Send Layer* (MSL) and *View Management Layer* (VML) at each process. In the figure, FD denotes any failure detector module satisfying the abstract properties for class $\diamond\tilde{\mathcal{P}}$ as defined in Sect. 4.

All interactions with the communication network and the failure detector are limited to MSL which uses the unreliable, unsequenced datagram transport

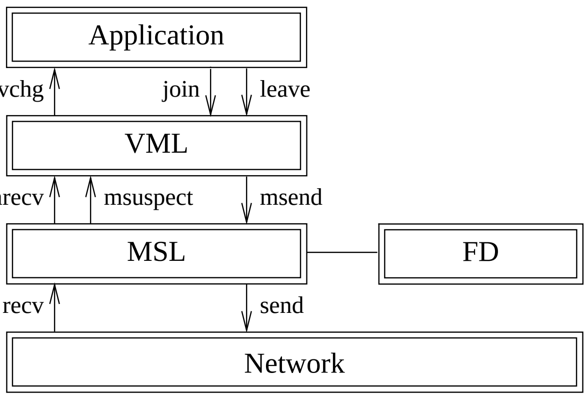

Fig. 1. Overall structure of the partitionable group membership service.

service of the network through the primitives *send*() and *recv*(). Each MSL can also read the suspect list of the corresponding failure detector module FD. M-SL implements the primitives *msend*(), *mrecv*() and *msuspect*() as described below, which in turn are used by VML. Recall that we consider group membership changes due to failures and repairs only. Thus, the implementation we give includes only the view change notification event *vchg*() but not the primitives *join*() and *leave*() for voluntarily joining and leaving the group.

In order to distinguish between the various layers in our discussion, we say that a process *m-sends* and *m-receives* messages when it communicates through the MSL primitives *msend*() and *mrecv*(), respectively. We reserve *send* and *receive* to denote communication directly through the network services without going through MSL. Similarly, we say that a process *m-suspects* those processes that are notified through a *msuspect*() event while *suspect* is reserved for describing the failure detector itself.

The following notation is used in the presentation of our algorithms. We use *italic* font for variable and procedure names. Tags denoting message types are written in SMALLCAPS. The **wait-for** construct is used to block a process until a *mrecv*() or a *msuspect*() event is generated by MSL. The **generate** construct produces an upcall of the specified type to the next layer in the architecture.

5.1 The Multi-Send Layer

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 MSL is to hide this complexity by transforming the unreliable, point-to-point network communication primitives to their best-effort, one-to-many counterparts. Informally, MSL tries to deliver m-sent messages to all processes in some destination set. MSL 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 MSL corresponds to the transitive closure of reachability at the failure detector layer. What distinguishes MSL from a typical network routing or reliable multicast service is the integration of message delivery semantics with the reachability information. In that sense, MSL is much closer to the *dynamic routing layer* of Phoenix [24] and the MUTS layer of Horus [34].

Informally, properties that MSL must satisfy are:

Property 3. (a) if a process q is continuously unreachable from p, then eventually p will continuously m-suspect q; (b) if a process q is continuously reachable from p, then eventually every process that m-suspects q also m-suspects p; (c) each process m-receives a message at most once and only if some process actually m-sent it earlier; (d) messages from the same sender are m-received in FIFO order; (e) a message that is m-sent by a correct process is eventually m-received by all processes in the destination set that are not m-suspected; (f) a process never m-suspects itself; (g) the reachability relation defined by the *msuspect*() events is eventually symmetric.

Properties (a) and (b) are the non-triviality conditions of our communication service. Properties (c) and (d) place simple integrity and order requirements on m-receiving messages. Property (e) defines a liveness condition on the m-sending of messages. Finally, property (f) prevents processes from m-suspecting themselves, while property (g) requires that if a correct process p stops m-suspecting another correct process q, then eventually q will stop m-suspecting p. It is important to note that from the combination of properties (b) and (f) we conclude that if q is continuously reachable from p, then p eventually stops m-suspecting q. Moreover, from properties (b) and (e) we conclude that if q is continuously reachable from p, then every message m-sent by q to p is eventually m-received.

A formal description of these properties, along with an algorithm to achieve them can be found in the extended version of this work [6]. The proposed algorithm is based on the integration of a routing algorithm and a failure detector of class $\diamond\tilde{\mathcal{P}}$.

5.2 The View Management Layer

VML uses the services provided by MSL in order to construct and install views as defined by the PGMS specification. At each process, let the *reachable set* correspond to those processes that are not currently m-suspected. These reachable sets form a good basis for constructing views since part of the PGMS specification follows immediately from the properties of MSL that produce them. In particular, Property GM2 is satisfied by Property 3(a) requiring that if a process q is continuously unreachable from p, then eventually p will continuously m-suspect q. Property GM1 is satisfied by Properties 3(b) and 3(f), as discussed above. Finally, Property GM5 is satisfied by Property 3(f).

The main difference between reachable sets as constructed by MSL and views as defined by PGMS is with respect to coherency. While reachable sets are

```
thread $ViewManagement$
    $reachable \leftarrow \{p\}$                  % Set of unsuspected processes
    $version \leftarrow (0, \ldots, 0)$           % Vector clock
    $symset \leftarrow (\{p\}, \ldots, \{p\})$    % Symmetry set
    $view \leftarrow (\ UniqueID(),\ \{p\}\ )$    % Current view id and composition
    $cview \leftarrow view$                       % Corresponding complete view
    generate $vchg(view))$

    while true do
        wait-for $event$                          % Remain idle until some event occurs
        case $event$ of

            $msuspect(P)$:
                foreach $r \in (\Pi - P) - reachable$ do $symset[r] \leftarrow reachable$
                $msend(\langle \text{SYMMETRY}, version, reachable \rangle, (\Pi - P) - reachable)$
                $reachable \leftarrow \Pi - P$
                $AgreementPhase()$

            $mrecv(\langle \text{SYNCHRONIZE}, V_p, V_q, P \rangle, q)$:
                if $(version[q] < V[q])$ then
                    $version[q] \leftarrow V[q]$
                    if $(q \in reachable)$ then
                        $AgreementPhase()$
                fi

        esac
    od
```

Fig. 2. The main algorithm for process p.

completely individualistic and lack any coordination, views of different processes need to be coherent among themselves as defined by Property GM3. VML achieves this property by using reachable sets as initial estimates for new views but installs them only after having reached agreement on their composition among mutually-reachable processes. To guarantee liveness of our solution, each execution of the agreement algorithm must terminate by actually installing a new view. Yet the composition of installed views cannot invalidate any of the properties that are inherited from MSL as described above.

The main algorithm for VML, illustrated in Fig. 2, alternates between an *idle phase* and an *agreement phase*. A process remains idle until either it is informed by MSL that there is a change in its perception of the reachable set (through a *msuspect*() event), or it m-receives a message from another process that has observed such a change. Both of these events cause the process to enter agreement phase. The agreement protocol, illustrated in Fig. 3, is organized as two sub-phases called *synchronization phase* and *estimate exchange phase* (for short, s-phase and ee-phase, respectively).

At the beginning of s-phase, each process m-sends a synchronization message containing a version number to those processes it perceives as being reachable, and then waits for responses. This message acts to "wake-up" processes that have not yet entered s-phase. Furthermore, version numbers exchanged in the s-phase are used in subsequent ee-phases to distinguish between messages of different agreement protocol invocations. A process leaves s-phase to enter ee-phase either when it m-receives a response to its synchronization message from every process that has not been m-suspected during the s-phase, or when it m-receives a message from a process that has already entered ee-phase.

Each process enters ee-phase (Fig. 4 and 5) 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 approximation for reachability that is being reported to it by MSL. In order to guarantee liveness, the algorithm constructs estimate sets that are always monotone decreasing so that the agreement condition is certain to hold eventually. Whenever the estimate changes, the process m-sends a message containing the new estimate to every process belonging to the estimate itself. When a process m-receives estimate messages, it removes from its own 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 a process selected among the current estimate to act as a *coordinator*. Note that while estimates are evolving, different processes may select different coordinators. Or, the coordinator may crash or become unreachable 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 that proposals m-received from some set S of processes are all equal to S, agreement is achieved and the coordinator m-sends to the members of S a message containing a new view identifier and composition equal to S. When a process m-receives such a message, there are two possibilities: it can either install a *complete* view, containing all processes indicated in the message, or it can install a *partial* view, containing a subset of the processes indicated in the message. Partial views are necessary whenever the installation of a complete view would violate Property GM3(iii). This condition is checked by verifying whether the current views of processes composing the new view intersect[3]. If they do, this could mean that a process in the intersection has never installed one of the intersecting views, thus violating Property GM3(iii). For this reason, the m-received view is broken in to a set of non-intersecting partial views, each of them satisfying Property GM3(iii). If, on the other hand, current views do not intersect, each process can install the new complete view as m-received from the coordinator. Note that classification of views as being complete or partial is completely internal to the implementation. An application programmer using the provided service in unaware of the distinction and deals with a single notion of view. Although each invocation of the agreement protocol

[3] This condition can be checked locally since current views of processes composing the new view are included in the message from the coordinator.

```
**procedure** $\mathit{AgreementPhase}()$
    **repeat**
        $\mathit{estimate} \leftarrow \mathit{reachable}$                % Next view estimation
        $\mathit{version}[p] \leftarrow \mathit{version}[p] + 1$        % Generate new version number
        $\mathit{SynchronizationPhase}()$
        $\mathit{EstimateExchangePhase}()$
    **until** $\mathit{stable}$                              % Exit when the view is stable

**procedure** $\mathit{SynchronizationPhase}()$
    $\mathit{synchronized} \leftarrow \{p\}$                       % Processes syncronized with $p$
    **foreach** $r \in \mathit{estimate} - \{p\}$ **do**
        $\mathit{msend}(\langle \text{SYNCHRONIZE}, \mathit{version}[r], \mathit{version}[p], \mathit{symset}[r] \rangle, \{r\})$
    **while** $(\mathit{estimate} \not\subseteq \mathit{synchronized})$ **do**
        **wait-for** $\mathit{event}$                       % Remain idle until some event occurs
        **case** $\mathit{event}$ **of**

            $\mathit{msuspect}(P)$:
                **foreach** $r \in (\Pi - P) - \mathit{reachable})$ **do** $\mathit{symset}[r] \leftarrow \mathit{reachable}$
                $\mathit{msend}(\langle \text{SYMMETRY}, \mathit{version}, \mathit{reachable} \rangle, (\Pi - P) - \mathit{reachable})$
                $\mathit{reachable} \leftarrow \Pi - P$
                $\mathit{estimate} \leftarrow \mathit{estimate} \cap \mathit{reachable}$

            $\mathit{mrecv}(\langle \text{SYMMETRY}, V, P \rangle, q)$:
                **if** $(\mathit{version}[p] = V[p])$ **and** $(q \in \mathit{estimate})$ **then**
                    $\mathit{estimate} \leftarrow \mathit{estimate} - P$

            $\mathit{mrecv}(\langle \text{SYNCHRONIZE}, V_p, V_q, P \rangle, q)$:
                **if** $(\mathit{version}[p] = V_p)$ **then**
                    $\mathit{synchronized} \leftarrow \mathit{synchronized} \cup \{q\}$
                **if** $(\mathit{version}[q] < V_q)$ **then**
                    $\mathit{version}[q] \leftarrow V_q$
                    $\mathit{agreed}[q] \leftarrow V_q$
                    $\mathit{msend}(\langle \text{SYNCHRONIZE}, \mathit{version}[q], \mathit{version}[p], \mathit{symset}[q] \rangle, \{q\})$
                **fi**

            $\mathit{mrecv}(\langle \text{ESTIMATE}, V, P \rangle, q)$
                $\mathit{version}[q] = V[q]$
                **if** $(q \notin \mathit{estimate})$ **then**
                    $\mathit{msend}(\langle \text{SYMMETRY}, \mathit{version}, \mathit{estimate} \rangle, \{q\}$
                **elseif** $(\mathit{version}[p] = V[p])$ **and** $(p \in P)$ **then**
                    $\mathit{estimate} \leftarrow \mathit{estimate} \cap P$
                    $\mathit{synchronized} \leftarrow P$
                    $\mathit{agreed} \leftarrow V$
                **fi**

        **esac**
    **od**
```

Fig. 3. Agreement and synchronization phases for process p.

terminates with the installation of a new view, it is possible that the new view does not correspond to the current set of processes perceived as being reachable, or that a new synchronization message has been m-received during the previous ee-phase. In both cases a new agreement phase is started.

In the extended version of this work [6], we give a proof that our algorithm satisfies the properties of PGMS. Here, we discuss in high-level terms the techniques used by the algorithm in order to satisfy the specification. Leaving out the more trivial properties such as View Completeness, View Integrity and View Order, we focus our attention on View Coherency, View Accuracy and liveness of the solution. Property GM3 consists of three parts. GM3(iii) is satisfied through the installation of partial views, as explained above. As for GM3(i) and GM3(ii), each process is eventually informed if another process belonging to its current view has not installed it, or if it has changed views after having installed it, respectively. The process is kept informed either through a message, or through an m-suspect event. In both cases, it reenters the agreement phase. As for liveness, each invocation of the agreement protocol terminates with the installation of a new view, even in situations where the reachability relation is highly unstable. 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 MSL never result in a process being 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. In the limit, the estimate set will eventually reduce to the process itself and a singleton view will be installed. This approach may seem in conflict with View Accuracy: if process p m-receives from process r a message inviting it to remove a process q, it cannot refuse it. But if p and q are permanently reachable, non-triviality properties of MSL guarantee that after some time, r cannot remove q from its view estimate without removing p as well. So, after some time, r cannot m-send a message to p inviting it to exclude q, because p cannot belong to the current estimate of r. Moreover, s-phase of view agreement constitutes a "barrier" against the propagation of old "remove q" messages. In this way, it is possible to show that there is a time after which all views installed by p contain q.

A more detailed description of the VML algorithm can be found in the extended version of this work [6].

6 Reliable Multicast Service: Specification

The class of partition-aware applications that can be programmed using group membership alone is limited [7]. In general, network applications require closer cooperation that is facilitated through communication among group members. In this section, we briefly illustrate how the group membership service of Sect. 3 may constitute the basis of more complex group communication services. The

```
procedure $EstimateExchangePhase()$
    $installed \leftarrow$ false                     % True when a new view is installed
    $InitializeEstimatePhase()$
    repeat
        wait-for $event$                         % Remain idle until some event occurs
        case $event$ of

            $msuspect(P)$:
                foreach $r \in (\Pi - P) - reachable$ do $symset[r] \leftarrow reachable$
                $msend(\langle \text{SYMMETRY}, version, reachable\rangle, (\Pi - P) - reachable)$
                $msend(\langle \text{ESTIMATE}, agreed, estimate\rangle, (\Pi - P) - reachable)$
                $reachable \leftarrow \Pi - P$
                if $(estimate \cap P \neq \emptyset)$ then
                    $SendEstimate(estimate \cap P)$

            $mrecv(\langle \text{SYMMETRY}, V, P\rangle, q)$:
                if $(agreed[p] = V[p]$ or $agreed[q] \leq V[q])$ and $(q \in estimate)$ then
                    $SendEstimate(estimate \cap P)$

            $mrecv(\langle \text{SYNCHRONIZE}, V_p, V_q, P\rangle, q)$:
                $version[q] \leftarrow V_q$
                if $(agreed[q] < V_q)$ and $(q \in estimate)$ then
                    $SendEstimate(estimate \cap P)$

            $mrecv(\langle \text{ESTIMATE}, V, P\rangle, q)$:
                if $(q \in estimate)$ then
                    if $(p \notin P)$ and $(agreed[p] = V[p]$ or $agreed[q] \leq V[q])$ then
                        $SendEstimate(estimate \cap P)$
                    elseif $(p \in P)$ and $(\forall r \in estimate \cap P : agreed[r] = V[r])$ then
                        $SendEstimate(estimate - P)$
                fi

            $mrecv(\langle \text{PROPOSE}, S\rangle, q)$:
                $ctbl[q] \leftarrow S$
                if $(q \in estimate)$ and $CheckAgreement(ctbl)$ then
                    $InstallView(UniqueID(), ctbl)$
                    $installed \leftarrow$ true
                fi

            $mrecv(\langle \text{VIEW}, w, C\rangle, q)$:
                if $(C[p].cview.id = cview.id)$ and $(q \in estimate))$ then
                    $InstallView(w, C)$
                    $installed \leftarrow$ true
                fi

        esac
    until $installed$
```

Fig. 4. Estimate exchange phase for process p: Part (a).

```
procedure $\mathit{InitializeEstimatePhase}()$
    $\mathit{SendEstimate}(\emptyset)$

procedure $\mathit{SendEstimate}(P)$
    $\mathit{estimate} \leftarrow \mathit{estimate} - P$
    $\mathit{msend}(\langle \text{ESTIMATE}, \mathit{agreed}, \mathit{estimate}\rangle, \mathit{reachable} - \{p\})$
    $\mathit{msend}(\langle \text{PROPOSE}, (\mathit{cview}, \mathit{agreed}, \mathit{estimate})\rangle, \mathit{Min}(\mathit{estimate}))$

function $\mathit{CheckAgreement}(C)$
    return  $(\forall q \in C[p].\mathit{estimate} : C[p].\mathit{estimate} = C[q].\mathit{estimate})$
            and $(\forall q, r \in C[p].\mathit{estimate} : C[p].\mathit{agreed}[r] = C[q].\mathit{agreed}[r])$

procedure $\mathit{InstallView}(w, C)$
    $\mathit{msend}(\langle \text{VIEW}, w, C\rangle, C[p].\mathit{estimate} - \{p\})$
    if $(\exists q, r \in C[p].\mathit{estimate}$ :
            $q \in C[r].\mathit{cview}.\mathit{comp} \wedge C[q].\mathit{cview}.\mathit{id} \neq C[r].\mathit{cview}.\mathit{id})$ then
        $\mathit{view} \leftarrow (\ (w, \mathit{view}.\mathit{id}),\ \{r | r \in C[p].\mathit{estimate} \wedge C[r].\mathit{cview}.\mathit{id} = \mathit{cview}.\mathit{id}\})$
    else
        $\mathit{view} \leftarrow ((w, \perp), C[p].\mathit{estimate})$
    generate $\mathit{vchg}(\mathit{view})$
    $\mathit{cview} \leftarrow (w, C[p].\mathit{estimate})$
    $\mathit{stable} \leftarrow (\mathit{view}.\mathit{comp} = \mathit{reachable})$ and
            $(\forall q, r \in C[p].\mathit{estimate} : C[p].\mathit{agreed}[r] = \mathit{agreed}[r])$
```

Fig. 5. Estimate exchange phase for process p: Part (b).

proposed extension is based on a reliable multicast service with *view synchrony* semantics that governs the delivery of multicast messages with respect to installation of views. After having introduced the reliable multicast specification, we illustrate how our solution for PGMS may be easily extended in order to implement view synchrony.

Group members communicate through reliable multicasts by invoking the primitive $\mathit{mcast}(m)$ that attempts to deliver message m to each of the processes in the current view through a $\mathit{dlvr}()$ upcall. Multicast messages are labeled in order to be globally unique. To simplify the presentation, we use M_p^v to denote the set of messages that have been delivered by process p in view v.

Ideally, all correct processes belonging to a given view should deliver the same set of messages in that view. In partitionable systems, this requirement could result in the multicast to block until communication failures are repaired so that those processes that have become unreachable since the view was installed can deliver the message. Thus, we relax this condition on the delivery of messages as follows: a process q may be exempt from delivering the same set of messages as some other correct process p in a view v if q crashes or if it becomes unreachable from p. In other words, agreement on the set of delivered messages in a view v is limited to those processes that survive a view change from view v to the same next view.

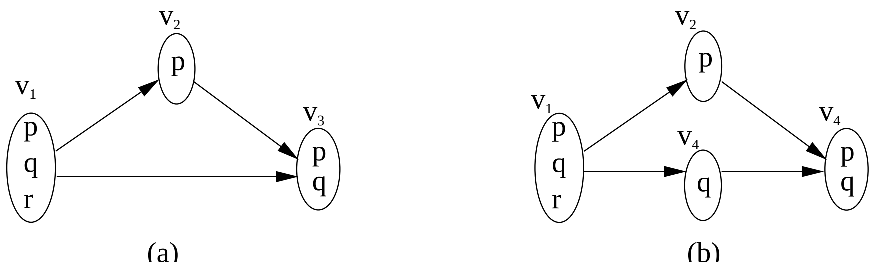

Fig. 6. Merging scenarios. Ovals depict view compositions as sets of process names. Directed edges depict *immediate successor* relations between views.

RM1 (Message Agreement). *Given two views v and w such that w is an immediate successor of v, all processes belonging to both views deliver the same set of multicast messages in view v. Formally,*

$$v \prec_p w \wedge q \in \overline{v} \cap \overline{w} \;\Rightarrow\; M_p^v = M_q^v \;.$$

The importance of Message Agreement can be better understood when considered together the properties offered by the group membership service specified in Sect. 3. Given two permanently reachable processes, there is a time after which they install the same sequence of views and deliver the same set of messages between every pair of successive views.

Note that Property RM1 places no restrictions on the set of messages delivered by a process q that belonged to view v along with p but that subsequently ends up in a different partition and is excluded from w. In this case, process q may or may not deliver some message m that was delivered by p in view v. If, however, q indeed delivers message m, it must do it in the same view v as p. This observation leads to the next property.

RM2 (Uniqueness). *Each multicast message, if delivered at all, is delivered in exactly one view. Formally,*

$$(m \in M_p^v) \wedge (m \in M_q^w) \;\Rightarrow\; v = w \;.$$

Properties RM1 and RM2 together define what has been called *view synchrony* in group communication systems. In distributed application development, view synchrony is extremely valuable since it admits global reasoning using local information only: Process p knows that all other processes surviving a view change along with it have delivered the same set of messages in the same view as p itself. And if two processes share some global state in a view and this state depends only on the set of delivered messages regardless of their order, then they will continue to share the same state in the next view if they both survive the view change[4].

[4] For applications where the shared state is sensitive to the order in which messages are delivered, specific order properties can be enforced by additional system layers.

Unfortunately, the group communication service specified so far does not allow global reasoning based on local information in partitionable systems. Consider the scenario depicted in Fig. 6(a) where three processes p, q, r have all installed view v_1. At some point process r crashes and p becomes temporarily unreachable from q. Process p reacts to both events by installing view v_2 containing only itself before merging back with q and installing view v_3. Process q, on other hand, reacts only to the crash of r and installs view v_3 excluding r. Suppose that p and q share the same state in view v_1 and that p modifies its state during v_2. When p and q install v_3, p knows immediately that their states may have diverged, while q cannot infer this fact based on local information alone. Therefore, q could behave inconsistently with respect to p. In an effort to avoid this situation, p could collaborate by sending q a warning message as soon as it installs view v_3, but q could perform inconsistent operations before receiving such a message. The problem stems from the fact that views v_1 and v_2, that merge to form view v_3, have at least one common member (p). The scenario of the above example can be easily generalized to any run where two overlapping views merge to form a common view. We rule out these runs with the following property.

RM3 (Merging Rule). *Two views merging into a common view must have disjoint compositions. Formally,*

$$(v \prec u) \wedge (w \prec u) \wedge (v \neq w) \;\Rightarrow\; \overline{v} \cap \overline{w} = \emptyset \enspace .$$

The sequence of view installations in a run respecting this property is shown in Fig. 6(b): Before installing v_3, process q has to first install view v_4. Thus the two views that merge to form v_3 have empty intersection. As a result, when p and q install view v_3, they both knows immediately that their states could have diverged during the partitioning. Note that Property RM3 may appear to be part of the group membership specification since it is concerned with view installations alone. Nevertheless, we choose to include it as part of the reliable multicast service specification since RM3 becomes relevant only in the context of multicast message deliveries. In other words, applications that need no guarantees for multicast messages but rely on PGMS alone would not be interested in RM3.

The next property places simple integrity requirements on delivery of messages to prevent the same message from being delivered multiple times by the same process or a message from being delivered "out of thin air" without first being multicast.

RM4 (Message Integrity). *Each process delivers a message at most once and only if some process actually multicast it earlier. Formally,*

$$\sigma(p,t) = \mathit{dlvr}(m) \;\Rightarrow$$
$$(\mathit{dlvr}(m) \notin \sigma(p, \mathcal{T} - \{t\})) \wedge (\exists q, \exists t' < t : \sigma(q,t') = \mathit{mcast}(m)) \enspace .$$

Note that a specification consisting of Properties RM1–RM4 alone can be trivially satisfied by not delivering any messages at all. We exclude such useless solutions by including the following property.

RM5 (Liveness).
(i) A correct process always delivers its own multicast messages. Formally,

$$p \in \mathit{Correct}(C) \wedge \sigma(p,t) = \mathit{mcast}(m) \;\Rightarrow\; (\exists t' > t : \sigma(p,t') = \mathit{dlvr}(m)) \enspace .$$

(ii) Let p be a correct process that delivers message m in view v that includes some other process q. If q never delivers m, then p will eventually install a new view w as the immediate successor to v. Formally,

$$\begin{gathered} p \in \mathit{Correct}(C) \wedge \sigma(p,t) = \mathit{dlvr}(m) \wedge q \in \overline{v} \wedge \\ \mathit{view}(p,t) = v \wedge \mathit{dlvr}(m) \notin \sigma(q,\mathcal{T}) \;\Rightarrow \\ \exists t' > t : \sigma(p,t') = \mathit{vchg}(w) \enspace . \end{gathered} \qquad .$$

The second part of Property RM5 is the liveness counterpart of Property RM1: If a process p delivers a message m in view v containing some other process q, then either q also delivers m, or p eventually excludes q from its current view.

Properties RM1–RM5 that define our Reliable Multicast Service can be combined with Properties GM1–GM5 of group membership to obtain what we call a *Partitionable Group Communication Service* with *view synchrony* semantics. In the extended version of this work [6] we show how our solution to PGMS can be extended to satisfy this specification.

7 Related Work and Discussion

The process group paradigm has been the focus of extensive experimental work in recent years and group communication services are gradually finding their way into systems for supporting fault-tolerant distributed applications. Examples of experimental group communication services include Isis [9], Transis [13], Totem [28], Newtop [17], Horus [35], Ensemble [21], Spread [2], Moshe [4] and Jgroup [27]. There have also been several specifications for group membership and group communication not related to any specific experimental system [30, 18, 31].

Despite this intense activity, the distributed systems community has yet to agree on a formal definition of the group membership problem, especially for partitionable systems. The fact that many attempts have been show to either admit trivial solutions or to exhibit undesirable behavior is partially responsible for this situation [3]. Since the work of Anceaume *et al.*, several other group membership specifications have appeared [20, 18, 4].

Friedman and Van Renesse [20] give a specification for the Horus group communication system that has many similarities to our proposal, particularly with respect to safety properties such as View Coherency and Message Agreement. There are, however, important differences with respect to non-triviality properties: The Horus specification is conditional on the outputs produced by a failure detector present in the system. This approach is also suggested by Anceaume *et al.* [3] and adopted in the work of Neiger [30]. We feel that a specification for group membership should be formulated based on properties of *runs* characterizing actual executions and not in terms of suspicions that a failure detector produces. Otherwise, the validity of the specification itself would be conditional on the properties of the failure detector producing the suspicions. For example, referring to a failure that never suspects anyone or one that always suspects everyone would lead to specifications that are useless. Thus, it is reasonable for the correctness of a group membership service *implementation*, but not its *specification*, to rely on the properties of the failure detector that it is based on.

Congress and Moshe [4] are two membership protocols that have been designed by the Transis group. Congress provides a simple group membership protocol, while Moshe extends Congress to provide a full group communication service. The specification of Moshe has many similarities with our proposal and includes properties such as *View Identifier Local Monotony*, *Self Inclusion* and *View Synchrony*, that can be compared to GM4, GM5 and RM1 of our proposal. Property RM5 is implied by Properties *Self Delivery* and *Termination of Delivery* of Moshe. On the other hand, the specification of Moshe does not guarantee Properties RM3 and RM4, thus undesirable scenarios similar to those described in Sect. 6 are possible. The main differences between Moshe and our proposal are with respect to non-triviality requirements. Moshe includes a property called *Agreement on Views* that may be compared to our Properties GM1, GM2 and GM3. The *Agreement on Views* property forces a set of processes, say S, to install the same sequence of views only if there is a time after which every process in S (i) is correct, (ii) is mutually reachable from all other processes in S, (iii) is mutually unreachable from all processes not in S, and (iv) is not suspected by any process in S. As in our proposal, this requirement may be relaxed by requiring that the condition hold only for a sufficiently long period of time, and not forever. Despite these similarities, the non-triviality requirements of Moshe and our proposal have rather different implications. For example, Moshe does not guarantee that two processes will install a common sequence of views even if they are mutually and permanently reachable but there are other processes in the system that become alternately reachable and unreachable from them. In our proposal, however, processes that are mutually and permanently reachable always install the same sequence of views, regardless of the state of the rest of the system. And this is desirable since a common sequence of installed views which is the basis for consistent collaboration in our partition-aware application development methodology.

Fekete *et al.* present a formal specification for a partitionable group communication service [18]. In the same work, the service is used to construct an ordered broadcast application and, in a subsequent work, to construct replicated data services [23]. The specification separates safety requirements from performance and fault-tolerance requirements, which are shown to hold in executions that stabilize to a situation where the failure status stops changing. The basic premise of Fekete *et al.* is that existing specifications for partitionable group communication services are too complex, thus, unusable by application programmers. And they set out to devise a much simpler formal specification, crafted to support the specific application they have in mind.

Simple specifications for partitionable group communication are possible only if services based on them are to support simple applications. Unfortunately, system services that are indeed useful for a wide range of applications are inherently more complex and do not admit simple specifications. Our experience in developing actual realistic partition-aware applications supports this claim [7].

The specification and implementation presented in this work form the basis of Jgroup [27], a group-enhanced extension to the Java RMI distributed object model. Jgroup enables the creation of object groups that collaborate using the facilities offered by our partitionable group communication service. Clients access an object group using the standard Java remote method invocation semantics and remotely invoking its methods as if it were a single, non-replicated remote object. The Jgroup system includes a *dependable registry service*, which itself is a partition-aware application built using Jgroup services [26]. The dependable registry is a distributed object repository used by object groups to advertise their services under a symbolic name (*register* operation), and by clients to locate object groups by name (*lookup* operation). Each registry replica maintains a database of bindings from symbolic group names to group object composition. Replicas are kept consistent using group communication primitives offered by Jgroup. During a partitioning, different replicas of the dependable registry may diverge. Register operations from within a partition can be serviced as long as at least one replica is included inside the partition. A lookup, on the other hand, will not be able to retrieve bindings that have been registered outside the current partition. Nevertheless, all replicas contained within a given partition are kept consistent in the sense that they maintain the same set of bindings and behave as a non-replicated object. When partitions merge, a reconciliation protocol is executed to bring replicas that may have been updated in different partitions back to a consistent state. This behavior of the dependable registry is perfectly reasonable in a partitionable system where clients asking for remote services would be interested only in servers running in the same partition as themselves.

8 Conclusions

Partition-aware applications are characterized by their ability to continue operating in multiple concurrent partitions as long as they can reconfigure themselves consistently [7]. A group membership service provides the necessary properties so that this reconfiguration is possible and applications can dynamically establish which services and at what performance levels they can offer in each of the partitions. The *primary partition* version of group membership is not suitable for supporting partition-aware applications since progress would be limited to at most one network partition. In this paper we have given a formal specification for a partitionable group communication service that is suitable for supporting partition-aware applications. Our specification excludes trivial solutions and is free from undesirable behaviors exhibited by previous attempts. Moreover, it requires services based on it to be live in the sense that view installations and message deliveries cannot be delayed arbitrarily when conditions require them.

We have shown that our specification can be implemented in any asynchronous distributed system that admits a failure detector satisfying *Strong Completeness* and *Eventual Strong Accuracy* properties. The correctness of the implementation depends solely on these abstract properties of the failure detector and not on the operating characteristics of the system. Any practical failure detector implementation presents a trade-off between accuracy and responsiveness to failures. By increasing acceptable message delays after each false suspicion, accuracy can be improved but responsiveness will suffer. In practice, to guarantee reasonable responsiveness, finite bounds will have to be placed on acceptable message delays, perhaps established dynamically on a per channel or per application basis. Doing so will guarantee that new views will be installed within bounded delays after failures. This in turn may cause some reachable processes to be excluded from installed views. Such processes, however, have to be either very slow themselves or have very slow communication links, and thus, it is reasonable to exclude them from views until their delays return to acceptable levels.

Each property included in our specification has been carefully studied and its contribution evaluated. We have argued that excluding any one of the properties makes the resulting service either trivial, or subject to undesirable behaviors, or less useful as a basis for developing large classes of partition-aware applications. Specification of new system services is mostly a social process and "proving" the usefulness of any of the included properties is impossible. The best one can do is program a wide range of applications twice: once using a service with the proposed property, and a second time without it, and compare their relative difficulty and complexity. We have pursued this exercise for our specification by programming a set of practical partition-aware applications [7]. In fact, the specification was developed by iterating the exercise after modifying properties based on feedback from the development step. As additional empirical evidence in support of our specification, we point to the Jgroup system based entirely on the specification and implementation given in this paper. As discussed in Sect. 7, the dependable registry service that is an integral part of Jgroup has been programmed using services offered by Jgroup itself. Work is currently underway in using Jgroup to develop other partition-aware financial applications and a partitionable distributed version of the Sun tuple space system called Javaspaces.

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):311–342, November 1995.
2. Y. Amir and J. Stanton. The Spread Wide-Aread Group Communication System. Technical report, Center of Networking and Distributed Systems, Johns Hopkins University, Baltimore, Mariland, April 1998.

3. E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the Formal Specification of Group Membership Services. Technical Report TR95-1534, Department of Computer Science, Cornell University, August 1995.
4. T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable Group Membership Services for Novel Applications. In *Proceedings of the DIMACS Workshop on Networks in Distributed Computing*, pages 23–42. American Mathematical Society, 1998.
5. Ö. Babaoğlu, R. Davoli, L.A. Giachini, and M.G. Baker. RELACS: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICS)*, pages 612–621, Maui, Hawaii, January 1995.
6. Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. Technical Report UBLCS-98-1, Department of Computer Science, University of Bologna, April 1998.
7. Ö. Babaoğlu, R. Davoli, A. Montresor, and R. Segala. System Support for Partition-Aware Network Applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 184–191, Amsterdam, The Netherlands, May 1998.
8. K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, December 1993.
9. K. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Computer Society Press, 1994.
10. T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322–330, May 1996. Also available as technical report TR95-1533, Department of Computer Science, Cornell University.
11. T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(1):225–267, March 1996.
12. D. Dolev, R. Friedman, I. Keidar, and D. Malki. Failure Detectors in Omission Failure Environments. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, Santa Barbara, California, August 1997. Also available as Technical Report TR96-1608.
13. D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4), April 1996.
14. D. Dolev, D. Malki, and R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. Technical Report CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.
15. D. Dolev, D. Malki, and R. Strong. A Framework for Partitionable Membership Service. Technical Report CS95-4, Institute of Computer Science, The Hebrew University of Jerusalem, 1995.
16. D. Dolev, D. Malki, and R. Strong. A Framework for Partitionable Membership Service. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, May 1996.
17. P.E. Ezhilchelvan, R.A. Macêdo, and S.K. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS)*, Vancouver, BC, Canada, June 1995.
18. A. Fekete, N. Lynch, and A. Shvartsman. Specifying and Using a Partitionable Group Communication Service. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC)*, Santa Barbara, California, August 1997.

19. M.J. Fischer, N.A. Lynch, and M.S. Patterson. Impossibility of Distributed Consensus with one Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.
20. R. Friedman and R. Van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95-1537, Department of Computer Science, Cornell University, March 1995.
21. M. Hayden. *The Ensenble System*. PhD thesis, Department of Computer Science, Cornell University, January 1998.
22. F. Kaashoek and A. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *Proceedings of the 12th IEEE Symp. on Reliable Distributed Systems*, pages 222–230, Arlington, TX, May 1991.
23. R. Khazan, A. Fekete, and N. Lynch. Multicast Group Communication as a Base for a Load-Balancing Replicated Data Service. In *Proceedings of the 12th Symposium on Distributed Computing*, August 1998.
24. C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large-Scale Networks*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1996.
25. C. Malloth and A. Schiper. View Synchronous Communication in Large Scale Networks. In *Proceedings of the 2nd Open Workshop of the ESPRIT Project Broadcast*, Grenoble, France, July 1995.
26. A. Montresor. A Dependable Registry Service for the Jgroup Distributed Object Model. In *Proceedings of the 3rd European Reasearch Seminar on Advances in Distributed Systems (ERSADS)*, Madeira, Portugal, April 1999.
27. A. Montresor. The Jgroup Reliable Distributed Object Model. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Systems (DAIS)*, Helsinki, Finland, June 1999.
28. L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Group Communication System. *Communications of the ACM*, 39(4), April 1996.
29. L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agarwal. Extended Virtual Synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS)*, Poznan, Poland, June 1994.
30. G. Neiger. A New Look at Membership Services. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, May 1996.
31. R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A Dynamic View-Oriented Group Communication Service. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, June 1998.
32. A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 341–352, August 1991.
33. A. Schiper and A. Ricciardi. Virtually-synchronous Communication Based on a Weak Failure Suspector. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS)*, pages 534–543, June 1993.
34. R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus System. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, 1993.
35. R. van Renesse, K.P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.