Group Communication in Partitionable Systems: Specification and Algorithms

Özalp Babaoglu, Renzo Davoli, *Member*, *IEEE Computer Society*, and Alberto Montresor

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.

Index Terms—Group communication, view synchrony, partition-awareness, asynchronous systems, fault tolerance.

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, nonfunctional 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 nonfunctional 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 nonfunctional properties beyond certain levels. For this reason, nonfunctional 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 are 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 depend heavily on the application semantics; establishing them for arbitrary applications is beyond the scope of this paper. 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 [31], [21] that has traditionally characterized partitioned operation of network applications. In this paper, 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 [21], [7] 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 authors are with the Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy. E-mail: {babaoglu, davoli, montresor}@cs.unibo.it.*

Manuscript received 15 July, 1998; revised 1 June, 1999; accepted 31 Aug. 1999.
Recommended for acceptance by K. Marzullo.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107154.

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 [28], [1], [13], [14], [15], [16], [5], [32]. 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:

1. They are informal or ambiguous [32], [5], [15],
2. They cease to install new views even in cases where the group membership continues to change [16],
3. They capriciously split the group into several concurrent views, possibly down to singleton sets [28], [1], [13], [14], [16],
4. They capriciously install views without any justification from the operating environment [13], [14].

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 paper, we give a formal specification for partitionable group membership services that has the following desirable properties: 1) It does not suffer from any of the problems that have been observed for previous solutions; 2) It is implementable in asynchronous distributed systems that exhibit certain stability conditions which we formally characterize; 3) 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 [6]. For this reason, the current paper is limited to the specification of these services and their implementability.

The rest of the paper 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 Section 3, we give a formal specification for partitionable group membership services that guarantees liveness and excludes useless solutions. In Section 4, we extend the *failure detector* abstraction of Chandra and Toueg [10] to partitionable systems and show how it can be implemented in practical asynchronous systems where certain stability conditions hold. In Section 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 Section 8 concludes the work.

2 System Model

We adopt notation and terminology similar to that of Chandra and Toueg [10]. The system is comprised of a set Π of processes that can communicate by exchanging messages through a network. Processes are associated with 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 bound. 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 unbound 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 Section 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 $send(m,q)$ and receives a message m that has been sent to it by executing $recv(m)$. 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 (1) all messages sent are globally unique and (2) 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 $Correct(C) = \{p \mid \forall t : p \notin C(t)\}$ 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* [10], as we discuss in Section 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 paper, 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 2.1 (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 1) a process that has crashed by time t is unreachable from every other process at time t and 2) a process that has not crashed by time t is reachable from itself at time t. Formally,*[1]

$$1.\ p \in C(t) \Rightarrow q \not\rightsquigarrow_t p$$
$$2.\ p \notin C(t) \Rightarrow p \rightsquigarrow_t p$$

By definition, the unreachability pattern subsumes the crash pattern in every failure history. We nevertheless choose to model crash and unreachability patterns 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 2.2. (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 1) a crashed process stops executing events, and 2) 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,*

$$1.\ p \in C(t) \Rightarrow \forall t' \geq t : \sigma(p,t') = \epsilon$$
$$2.\ \sigma(p,t) = send(m,q) \Rightarrow (recv(m) \in \sigma(q,\mathcal{T}) \Leftrightarrow p \rightsquigarrow_t q\ )$$

Note that, by Definition 2.2, 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 Section 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 is 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

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.

require a form *eventual* symmetry, as stated in the next property:

Property 2.1 (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,*

$$\exists t_0, \forall t \geq t_0 : p \rightsquigarrow_t q \Rightarrow \exists t_1, \forall t \geq t_1 : q \rightsquigarrow_t p$$
$$\exists t_0, \forall t \geq t_0 : p \not\rightsquigarrow_t q \Rightarrow \exists t_1, \forall t \geq t_1 : q \not\rightsquigarrow_t p$$

This is 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 2.2 (Fair Channels). *Let p and q be two processes that are not permanently unreachable from each other. If p sends an unbound number of messages to q, then q will receive an unbound number of these messages. Formally,*

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

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 toward 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 is comprised of 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 into 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 the 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 $vchg(v)$, denoting a view change that installs view v. The *current view* of process p at time t is v, denoted $view(p, t) = v$, 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,*

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

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,*

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

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. The 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).

1. *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,*

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

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

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

3. *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,*

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

Returning to the above example, the current view of process p cannot remain v indefinitely as GM3.2 requires p to eventually install a new view. By assumption, q never installs another view after w. Thus, by GM3.1, the new view 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, guarantees 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, GM3.1 and

GM3.2, 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. GM3.3 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 deduce 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 also 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 2.2, 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 (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

Note that Properties GM1, GM2, and GM3 place constraints on the composition of the current of a process in relation to characteristics of the operating environment or actions of other processes. As such, they can be satisfied only by installing a new view whenever the current view of a process does not conform to the specification. In an asynchronous system, it is impossible to put a time bound on this action. Thus, the properties are stated so as to hold eventually. This is sufficient to guarantee that our specification of PGMS is *live* since it excludes implementations where installation of justified views are arbitrarily delayed.

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 arbitrarily delay installation of a new view, including processes that remain reachable for sufficiently long periods. Nor can it arbitrarily delay 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 [18] and primary partition group membership [9] are among them. Partitionable group membership service, as we have defined it, happens to be not solvable in an asynchronous system as well. Here, we give a proof sketch of this impossibility. Consider a system consisting of two processes p and q. Let $\mathcal{R}$ be the set of all runs in which p and q are correct and are permanently reachable from each other. Any implementation of our specification must guarantee that, for all runs in $\mathcal{R}$, there exists a time after which the current view of p always contains q. For contradiction, we will show that there is at least one run in $\mathcal{R}$ for which such a time does not exist. Let R_0 be a run in $\mathcal{R}$ and let t_0 be a time value. There are two possibilities: If the current view of p never contains q after t_0 in R_0, our claim is trivially proven. Thus, suppose the current view of p contains q at time t_0 in run R_0. Now consider another run R_0', identical to R_0 up to t_0, at which time process q crashes. By Property GM2, there must be a time $t_0' > t_0$ by which p installs a view v in R_0' that excludes q. Now, consider run $R_1 \in \mathcal{R}$ such that 1) the global and failure histories of R_1 are identical to those of R_0' up to time t_0; 2) the global and failure histories of R_1 restricted to p are identical to those of R_0' up to time t_0'; and 3) all messages sent by q after t_0 are received by p after t_0'. At time t_0', process p cannot distinguish run R_1 from R_0'. Thus, p installs view v at time t_0' in run R_1 just as it does in run R_0'. Again, there are two possibilities. If the current view of p never contains q after t_0' in R_1, our claim is proven. Otherwise, let $t_1 > t_0'$ be a time such that the current view of p contains q in

R_1 at time t_1. We repeat this construction now starting from run R_1 and time t_1, obtaining a run $R_2 \in \mathcal{R}$ in which p installs a view excluding q at time $t_1' > t_1$. By iterating the construction, we obtain a run $R \in \mathcal{R}$ in which the current view of p never contains q after a certain time or p installs an unbound number of views that exclude q.

This 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 [10]. Then, in Section 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 [24], [11]. The failure detector abstraction originally proposed by Chandra and Toueg [10] 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, which 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 [10]. 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 [10]:

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:*

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

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:*

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

Borrowing from Chandra and Toueg [10], failure detectors satisfying Strong Completeness and Eventual Strong Accuracy are called *eventually perfect* and their class denoted as $\diamond\tilde{P}$. 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 [10]. Informally, *weak* completeness and accuracy require the corresponding property to hold only for *some* pairs 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 a failure detector hierarchy for partitionable systems, as well as reductions between them, is beyond the scope of this paper, we make a few brief observations. In the 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 $\mathcal{D} \in \diamond\tilde{P}$ in partitionable asynchronous systems, just as it is impossible to implement a failure detector belonging to any of the classes $\diamond P$, $\diamond Q$, $\diamond S$, and $\diamond W$ in asynchronous systems with perfectly reliable communication [10]. In practice, however, asynchronous systems are expected to exhibit reasonable behavior and failure detectors for $\diamond\tilde{P}$ can indeed be implemented. For example, consider the following algorithm, which is similar to that of Chandra and Toueg [10], 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 [10].

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 $\diamond\tilde{P}$ in partitionable asynchronous systems. On the other hand, in an asynchronous system, it is possible for some process p to observe an unbound 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 $\diamond\tilde{P}$. 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 satisfying 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 Section 3 on partitionable asynchronous systems augmented with a failure detector of class $\diamond\tilde{P}$. 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 $\diamond\tilde{P}$ as defined in Section 4.

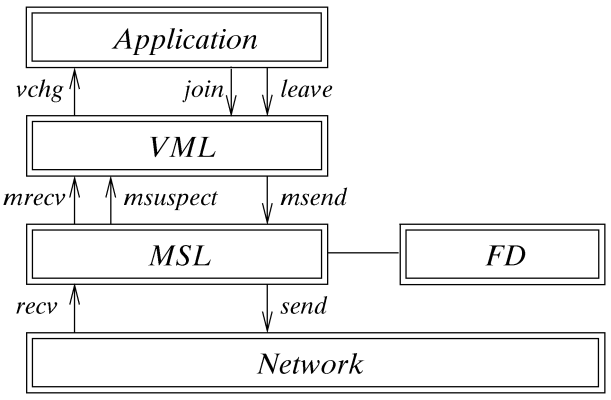


Fig. 1. Overall structure of the partitionable group membership service.

All interactions with the communication network and the failure detector are limited to MSL, which uses the unreliable, unsequenced datagram transport 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. MSL 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 an $msuspect()$ 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 an $mrecv()$ or an $msuspect()$ 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 into 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 [23] and the MUTS layer of Horus [33].

Informally, properties that MSL must satisfy are:

Property 5.1.

1. *If a process q is continuously unreachable from p, then eventually p will continuously m-suspect q;*
2. *If a process q is continuously reachable from p, then eventually every process that m-suspects q also m-suspects p;*
3. *Each process m-receives a message at most once and only if some process actually m-sent it earlier;*
4. *Messages from the same sender are m-received in FIFO order;*
5. *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;*
6. *A process never m-suspects itself;*
7. *The reachability relation defined by the $msuspect()$ events is eventually symmetric.*

Properties 5.1.1 and 5.1.2 are the nontriviality conditions of our communication service. Properties 5.1.3 and 5.1.4 place simple integrity and order requirements on m-receiving messages. Property 5.1.5 defines a liveness condition on the m-sending of messages. Finally, Property 5.1.6 prevents processes from m-suspecting themselves, while Property 5.1.7 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 5.1.2 and 5.1.6, we conclude that if q is continuously reachable from p, then p eventually stops m-suspecting q. Moreover, from Properties 5.1.2 and 5.1.5, 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 Appendix A. The proposed algorithm is based on the integration of a routing algorithm and a failure detector of class $\diamond\tilde{P}$.

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 5.1.1 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 5.1.1 and 5.1.6, as discussed above. Finally, Property GM5 is satisfied by Property 5.1.6.

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 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 the agreement phase. The agreement protocol, illustrated in Fig. 2, is organized as two subphases called *synchronization phase* and *estimate exchange phase* (for short, s-phase and ee-phase, respectively).

At the beginning of the 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 the 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 the s-phase to enter the 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 the ee-phase.

Each process enters the ee-phase (Fig. 4 and Fig. 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,

```
1   thread ViewManagement
2       reachable ← {p}                                   % Set of unsuspected processes
3       version ← (0, ..., 0)                             % Vector clock
4       symset ← ({p}, ..., {p})                          % Symmetry set
5       view ← ( UniqueID(), {p} )                        % Current view id and composition
6       cview ← view                                      % Corresponding complete view
7       generate vchg(view)
8
9       while true do
10          wait-for event                                % Remain idle until some event occurs
11          case event of
12
13              msuspect(P):
14                  foreach r ∈ (Π − P) − reachable do symset[r] ← reachable
15                  msend(⟨SYMMETRY, version, reachable⟩, (Π − P) − reachable)
16                  reachable ← Π − P
17                  AgreementPhase()
18
19              mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, q):
20                  if (version[q] < V[q]) then
21                      version[q] ← V[q]
22                      if (q ∈ reachable) then
23                          AgreementPhase()
24                  fi
25
26          esac
27      od
```

Fig. 2. The main algorithm for process p.

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 GM3.3. 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 GM3.3. For this reason, the m-received view is broken into a set of nonintersecting partial views, each of them satisfying Property GM3.3. 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 is unaware of the distinction and deals with a single notion of view. Although each invocation of the agreement protocol 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 Appendix D, 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. GM3.3 is satisfied through the installation of partial views, as explained above. As for GM3.1 and GM3.2, 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

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

```
1   procedure AgreementPhase()
2       repeat
3           estimate ← reachable                          % Next view estimation
4           version[p] ← version[p] + 1                   % Generate new version number
5           SynchronizationPhase()
6           EstimateExchangePhase()
7       until stable                                      % Exit when the view is stable
8
9
10  procedure SynchronizationPhase()
11      synchronized ← {p}                                % Processes with which p is syncronized
12      foreach r ∈ estimate − {p} do
13          msend(⟨SYNCHRONIZE, version[r], version[p], symset[r]⟩, {r})
14
15      while (estimate ⊈ synchronized) do
16          wait-for event                                % Remain idle until some event occurs
17          case event of
18
19              msuspect(P):
20                  foreach r ∈ (Π − P) − reachable) do symset[r] ← reachable
21                  msend(⟨SYMMETRY, version, reachable⟩, (Π − P) − reachable)
22                  reachable ← Π − P
23                  estimate ← estimate ∩ reachable
24
25              mrecv(⟨SYMMETRY, V, P⟩, q):
26                  if (version[p] = V[p]) and (q ∈ estimate) then
27                      estimate ← estimate − P
28
29              mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, q):
30                  if (version[p] = V_p) then
31                      synchronized ← synchronized ∪ {q}
32                  if (version[q] < V_q) then
33                      version[q] ← V_q
34                      agreed[q] ← V_q
35                      msend(⟨SYNCHRONIZE, version[q], version[p], symset[q]⟩, {q})
36                  fi
37
38              mrecv(⟨ESTIMATE, V, P⟩, q)
39                  version[q] = V[q]
40                  if (q ∉ estimate) then
41                      msend(⟨SYMMETRY, version, estimate⟩, {q}
42                  elseif (version[p] = V[p]) and (p ∈ P) then
43                      estimate ← estimate ∩ P
44                      synchronized ← P
45                      agreed ← V
46                  fi
47
48          esac
49      od
```

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

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, nontriviality 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. Furthermore, the 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 Appendix B.

5.3 Discussion

The main property of our algorithm is that the agreement protocol is guaranteed to terminate at correct processes, independent of the failure scenario. In particular, the

```
1   procedure EstimateExchangePhase()
2       installed ← false                                  % True when a new view is installed
3       InitializeEstimatePhase()
4
5       repeat
6           wait-for event                                 % Remain idle until some event occurs
7           case event of
8
9               msuspect(P):
10                  foreach r ∈ (Π − P) − reachable do symset[r] ← reachable
11                  msend(⟨SYMMETRY, version, reachable⟩, (Π − P) − reachable)
12                  msend(⟨ESTIMATE, agreed, estimate⟩, (Π − P) − reachable)
13                  reachable ← Π − P
14                  if (estimate ∩ P ≠ ∅) then
15                      SendEstimate(estimate ∩ P)
16
17              mrecv(⟨SYMMETRY, V, P⟩, q):
18                  if (agreed[p] = V[p] or agreed[q] ≤ V[q]) and (q ∈ estimate) then
19                      SendEstimate(estimate ∩ P)
20
21              mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, q):
22                  version[q] ← V_q
23                  if (agreed[q] < V_q) and (q ∈ estimate) then
24                      SendEstimate(estimate ∩ P)
25
26              mrecv(⟨ESTIMATE, V, P⟩, q):
27                  if (q ∈ estimate) then
28                      if (p ∉ P) and (agreed[p] = V[p] or agreed[q] ≤ V[q]) then
29                          SendEstimate(estimate ∩ P)
30                      elseif (p ∈ P) and (∀r ∈ estimate ∩ P : agreed[r] = V[r]) then
31                          SendEstimate(estimate − P)
32                  fi
33
34              mrecv(⟨PROPOSE, S⟩, q):
35                  ctbl[q] ← S
36                  if (q ∈ estimate) and CheckAgreement(ctbl) then
37                      InstallView(UniqueID(), ctbl)
38                      installed ← true
39                  fi
40
41              mrecv(⟨VIEW, w, C⟩, q):
42                  if (C[p].cview.id = cview.id) and (q ∈ estimate)) then
43                      InstallView(w, C)
44                      installed ← true
45                  fi
46
47          esac
48      until installed
```

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

algorithm has been designed to tolerate any number of failures and repairs that occur before and *during* the execution of the agreement protocol. This capability, however, implies degraded performance in case of complex failure scenarios. To illustrate the point, recall that processes can be added to a view estimate only at the beginning of an agreement phase. This implies that, when a set of processes initiate agreement towards a new view, this view may be already obsolete since some processes may cease to be m-suspected in the meantime. This in turn implies that a new agreement phase has to be started immediately after the termination of the current agreement. Furthermore, the designated coordinator may change several times during an agreement due to some process m-suspecting the coordinator. Every time the coordinator changes, each process has to m-send its proposal to the new coordinator and wait for a reply. Finally, the installation of partial views needed to satisfy Property GM3 may cause further invocations of the agreement protocol.

Although many optimizations are possible (for example, we could add a subprotocol to handle the installation of a complete view after the installations of partial views

```
1   procedure InitializeEstimatePhase()
2       SendEstimate(∅)
3
4   procedure SendEstimate(P)
5       estimate ← estimate − P
6       msend(⟨ESTIMATE, agreed, estimate⟩, reachable − {p})
7       msend(⟨PROPOSE, (cview, agreed, estimate)⟩, Min(estimate))
8
9   function CheckAgreement(C)
10      return    (∀q ∈ C[p].estimate : C[p].estimate = C[q].estimate)
11                and (∀q, r ∈ C[p].estimate : C[p].agreed[r] = C[q].agreed[r])
12
13  procedure InstallView(w, C)
14      msend(⟨VIEW, w, C⟩, C[p].estimate − {p})
15      if (∃q, r ∈ C[p].estimate : q ∈ C[r].cview.comp ∧ C[q].cview.id ≠ C[r].cview.id) then
16          view ← ( (w, view.id), {r | r ∈ C[p].estimate ∧ C[r].cview.id = cview.id} )
17      else
18          view ← ((w, ⊥), C[p].estimate)
19      generate vchg(view)
20      cview ← (w, C[p].estimate)
21      stable ← (view.comp = reachable) and (∀q, r ∈ C[p].estimate : C[p].agreed[r] = agreed[r])
```

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

without exiting ee-phase, thus saving the costs of repeated executions of the agreement protocol), we argue that, in common failure scenarios, the performance of our algorithm is reasonable. In particular, consider the scenario in which a single process q crashes. Eventually, every process will m-suspect q and enter the s-phase. At the beginning of the s-phase, each process m-sends a SYNCHRONIZE message and waits for a reply. Then, each process m-sends a proposal to the designated coordinator and a ESTIMATE message to every other process. In this scenario, ESTIMATE messages m-received from others will be ignored since all processes have the same view estimate (current view minus q, the crashed process). When the coordinator observes agreement on the m-received proposals, it m-sends a VIEW message to all. Hence, four end-to-end message delays are in general sufficient to exclude a crashed process from a view. The same delay analysis also characterizes partition merges or partitionings that result in the formation of disjoint partitions.

6 Reliable Multicast Service: Specification

The class of partition-aware-applications that can be programmed using group membership alone is limited [6]. 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 Section 3 may constitute the basis for more complex group communication services. The 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 $mcast(m)$ that attempts to deliver message m to each of the processes in the current view through a $dlvr()$ 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 a multicast to block if at least one process belonging to the view becomes unreachable before having had a chance to deliver the message. In that case, the multicast would terminate only upon repair of the communication failures. 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.

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 with the properties offered by the group membership service specified in Section 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 ended up in a

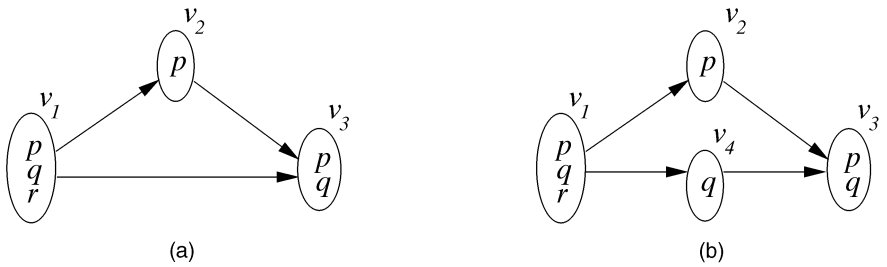


Fig. 6. Merging scenarios. Ovals depict view compositions as sets of process names. Directed edges depict *immediate successor* relations between views.

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]

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. 6a, 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, which 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,*

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

The sequence of view installations in a run respecting this property is shown in Fig. 6b: Before installing v_3, process q has to first install view v_4. Thus, the two views that merge to form v_3 have an empty intersection. As a result, when p and q install view v_3, they both know 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 RM.

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,*

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

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).

1. *A correct process always delivers its own multicast messages. Formally,*

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

2. *Let p be a correct process that delivers message m in view v that includes some other process q. If q never*

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.

delivers m, then p will eventually install a new view w as the immediate successor to v. Formally,

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

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 Appendix C, 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. Group communication services are gradually finding their way into systems for supporting fault-tolerant distributed applications. Examples of experimental group communication services include Isis [8], Transis [12], Totem [27], Newtop [16], Horus [34], Ensemble [20], Spread [2], Moshe [4], and Jgroup [26]. There have also been several specifications for group membership and group communication not related to any specific experimental system [29], [17], [30].

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 shown 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 [19], [17], [4].

Friedman and Van Renesse [19] 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 nontriviality 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 [29]. 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 on which it is based.

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 Moshe's Properties *Self Delivery* and *Termination of Delivery*. On the other hand, the specification of Moshe does not guarantee Properties RM3 and RM4, thus undesirable scenarios similar to those described in Section 6 are possible. The main differences between Moshe and our proposal are with respect to nontriviality 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 is

1. correct,
2. mutually reachable from all other processes in S,
3. mutually unreachable from all processes not in S, and
4. 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 nontriviality 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 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 [17]. In the same work, the service is used to construct an ordered broadcast application and, in a subsequent work, to construct replicated data services [22]. The specification separates safety requirements from performance and fault tolerance requirements, which are shown to hold in executions that stabilize 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. They set out to devise a more simple formal specification, crafted to support the specific application they have in mind. Quoting the authors:

> Our specification *VS* does not describe all the potentially useful properties of any particular implementation. Rather, it includes only the properties that are needed for the ordered broadcast application.

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 realistic partition-aware applications supports this claim [6].

The specification and implementation presented in this work form the basis of Jgroup [26], 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, nonreplicated remote object. The Jgroup system includes a *dependable registry service*, which itself is a partition-aware application built using Jgroup services [25]. 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 nonreplicated 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 [6]. 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 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 bound 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. 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, 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 [6]. 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 Section 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.

Appendix A

Multi-Send Layer: Specification and Implementation

In this appendix, we give a formal specification for MSL and an algorithm satisfying it. Recall that VML invokes the primitive $msend(m, G)$ of MSL for m-sending a message m to a destination set G. Messages are globally unique such that each message is m-sent at most once. MSL exports events $mrecv(m, q)$ and $msuspect(P)$ up to VML for m-receiving a

message m from process q and for m-suspecting processes in the set P, respectively. Note that $msuspect$ events are produced spontaneously by MSL and are not solicited explicitly by VML.

Let $\mathcal{R}(p,t)$ denote the *reachable set function*, defined as those processes that are not m-suspected by p at t. In other words, $q \in \mathcal{R}(p,t)$ if and only if the last $msuspect(P)$ event generated at process p by time t is such that $q \notin P$. Properties of MSL that are needed by the group membership algorithm are as follows.

Property A.1 (MSL Guarantees). *The Multi-Send Layer (MSL) satisfies the following properties: 1) If a process q is continuously unreachable from p, then eventually p will continuously m-suspect q; 2) if process q is continuously reachable from p, then eventually each process will m-suspect both or none of p and q; 3) each process m-receives a message at most once and only if some process actually m-sent it earlier; 4) messages from the same sender are m-received in FIFO order; 5) 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; 6) function $\mathcal{R}$ is perpetually reflexive; 7) function $\mathcal{R}$ is eventually symmetric. Formally,*

1.
$$\exists t_0, \forall p \in \Theta, \forall q \notin \Theta, \forall t \geq t_0 : p \not\rightsquigarrow_t q \Rightarrow$$
$$\exists t_1, \forall r \in Correct(C) \cap \Theta, \forall t \geq t_1 : \mathcal{R}(r,t) - \Theta = \emptyset.$$

2.
$$\exists t_0, \forall t \geq t_0 : p \rightsquigarrow_t q \Rightarrow \exists t_1, \forall t \geq t_1,$$
$$\forall r : p \in \mathcal{R}(r,t) \Leftrightarrow q \in \mathcal{R}(r,t).$$

3.
$$\sigma(p,t) = mrecv(m,q) \Rightarrow mrecv(m,q) \notin \sigma(p, \mathcal{T} - \{t\})$$
$$\wedge\, msend(m, G \cup \{p\}) \in \sigma(q, [0,t[).$$

4.
$$\sigma(p,t_1) = mrecv(m_1,q) \wedge \sigma(p,t_2) = mrecv(m_2,q)$$
$$\wedge\, t_1 < t_2 \Rightarrow \exists t_1', t_2' : t_1' < t_2' \wedge \sigma(q,t_1') = msend(m,G_1)$$
$$\wedge\, \sigma(q,t_2') = msend(m,G_2).$$

5.
$$\sigma(p,t) = msend(m, G \cup \{q\}) \Rightarrow \exists t' > t :$$
$$\sigma(q,t') = mrecv(m,p) \vee q \notin \mathcal{R}(p,t') \vee p \in C(t').$$

6.
$$p \in \mathcal{R}(p,t).$$

7.
$$p, q \in Correct(C), \exists t_0, \forall t \geq t_0 : q \in \mathcal{R}(p,t) \Rightarrow$$
$$\exists t_1, \forall t \geq t_1 : p \in \mathcal{R}(q,t).$$

In Fig. 7, we illustrate an algorithm for implementing MSL. Recall that our goal is simply to prove the implementability of the specification and not be concerned about efficiency. Thus, the algorithm uses a simple flooding strategy based on the forwarding of every received message on each output channel. At each process p, MSL maintains a local state defined by the variables $reachable$, seq, fd, ack, and msg. $reachable$ is the set of processes that are believed to be reachable through direct or indirect paths. This set is constructed from the outputs of the failure detector modules, including remote ones, as they are learned through incoming messages. seq is a vector indexed by Π, where $seq[q]$ represents the number of messages m-sent by p to q; it is used to generate the sequence numbers associated to messages. fd, ack, and msg are three vectors indexed by Π; for every process $q \in \Pi$, $fd[q]$ is a set of processes, $ack[q]$ is a vector (indexed by Π) of sequence numbers, while $msg[q]$ is a vector (indexed by Π) of sets of messages. The variables fd, ack, and msg may be seen as partitioned in two sections: The local section refers to process p ($fd[p]$, $ack[p]$, and $msg[p]$); the remote section refers to all other processes ($fd[q]$, $ack[q]$, and $msg[q]$ for each $q \neq p$). Process p modifies its local section in response to local events (for example, an $msend$ request from PGMS, the receipt of a message from the network, or a change in the output of the local failure detector module). In particular, $fd[p]$ records p's last reading of the failure detector, $ack[p][q]$ is the sequence number of the last message from q m-received by p, and $msg[p][q]$ is the set of messages m-sent by p to q for which p has not yet m-received an acknowledgment. For each process $q \neq p$, $fd[q]$, $ack[q]$, and $msg[q]$ contain p's local perception of the corresponding variables of q.

The algorithm is driven by $recv()$ events from below (network), $msend$ events from above (PGMS), and local $tick$ events that are produced periodically. It exports $mrecv$ and $msuspect$ events to PGMS. At each $tick$ event (lines 12-14), process p reads the output of the local failure detector module as $\mathcal{D}_p$ and sends a message containing the value of msg, ack, and fd to all other processes.

When p receives a message with contents $\langle M, A, F \rangle$ (lines 23-46), it verifies whether, for each process q, the data regarding q contained in the message are more recent than the ones saved in the local variables $msg[q]$, $ack[q]$, and $fd[q]$. The check is done through the predefined function $UpToDate()$. If so, p updates its local variables. If the set of messages sent by q to p as contained in M includes tuples of the form (SUSPECT, n, G) that have not yet been handled, then p temporarily m-suspects q and the processes in G by generating two $msuspect$ events, the first one m-suspecting the processes in G and the second one immediately removing the processes in G from the list of m-suspected processes. Then, p copies the value n in $ack[p][q]$ to indicate that the messages m-sent by q before m-suspecting p can be discarded. The SUSPECT tuples addressed to p by q are created to avoid persistent asymmetrical scenarios. The set G is chosen so that it guarantees property A.1.1. Then, p m-receives (in FIFO order) those messages that it has not yet done so. Finally, p modifies $msg[p]$ by removing all messages that have been acknowledged by q. At this point, p computes the transitive closure of the individual

```
1   thread MultiSend
2       seq ← (0, ..., 0)                                   % Sequence number array
3       ack ← ((0, ..., 0), ..., (0, ..., 0))              % Acknowledged messages
4       msg ← ( (∅, ..., ∅), ..., (∅, ..., ∅) )            % Messages to deliver
5       fd ← (∅, ..., ∅)                                    % Failure Detector readings
6       reachable ← {p}                                     % Reachable processes (directly or indirectly)
7
8       while true do
9           wait-for event                                  % Remain idle until some event occurs
10          case event of
11
12              tick:
13                  fd[p] ← D_p                             % Read the local failure detector output
14                  foreach q ∈ Π do send(⟨msg, ack, fd⟩, q)
15
16              msend(m, G):
17                  foreach q ∈ (G ∩ reachable) − {p} do
18                      seq[q] ← seq[q] + 1
19                      msg[p][q] ← msg[p][q] ∪ {(MSEND, m, seq[q])}
20                  od
21                  if (p ∈ G) then generate mrecv(m, p)
22
23              recv(⟨M, A, F⟩):
24                  foreach q ∈ Π − {p} : UpToDate(M[q], A[q], F[q]) do
25                      msg[q] ← M[q]; ack[q] ← A[q]; fd[q] ← F[q]
26                      if ((SUSPECT, n, G) ∈ M[q][p]) and (q ∈ reachable) then
27                          ack[p][q] ← n
28                          generate msuspect(Π − (reachable − G))
29                          generate msuspect(Π − reachable)
30                      fi
31                      foreach ( (MSEND, m, n) ∈ M[q][p] : n = ack[p][q] + 1 ) do
32                          ack[p][q] ← n
33                          generate mrecv(m, q)
34                      od
35                      msg[p][q] ← msg[p][q] − {(MSEND, −, n) | n < ack[p][q]}
36                  od
37                  P ← {p}; P' = ∅                         % Compute the transitive closure
38                  while (P ≠ P') do                       % of reachability
39                      P' ← P; P ← ⋃_{q∈P'} (Π − fd[q])
40                  foreach q ∉ P : (SUSPECT, −, −) ∉ msg[p][q] do
41                      seq[q] ← seq[q] + 1
42                      msg[p][q] ← {(SUSPECT, seq[q], P)}
43                  od
44                  if (P ≠ reachable) then
45                      generate msuspect(Π − P);
46                  reachable ← P
47
48          esac
49      od
```

Fig. 7. Algorithm for implementing MSL using failure detector $\mathcal{D} \in \diamond\tilde{P}$.

reachability sets. The resulting set P contains all the processes reachable from p through a direct or an indirect path. Finally, p creates the SUSPECT tuples needed to guarantee the eventual symmetry of function $\mathcal{R}$. If P differs from *reachable*, p generates the corresponding *msuspect* event and updates the set *reachable*.

To complete the algorithm explanation, a request to m-send message m to destination set G is handled in lines 16-21. For each q in the destination set G, the tuple (MSEND, m, n) is inserted into $msg[p][q]$ so that it will be sent to all processes at the next *tick* event. The value n represents the current sequence number. If p belongs to the destination set, then m is locally m-received. The MSEND tag is used to distinguish between messages sent on behalf of the upper layer and those sent internally which have SUSPECT tags.

We now prove that the algorithm of Fig. 7 is correct. Since we need to refer to several processes, we index

variable names with process names to which they are local (for example, $reachable_p$). Moreover, we denote variables as functions of time so as to refer to their value at a particular time (for example, $reachable_p(t)$).

Theorem A.1 (MSL Guarantees). *The algorithm of Fig. 7 satisfies MSL as specified in Property A.1.*

Proof.

1. We must show that if a process q is continuously unreachable from p, then eventually p will continuously m-suspect q. Let Θ and $\Psi = \Pi - \Theta$ be two sets of processes such that there is a time t_0 after which the processes in Ψ are permanently unreachable from processes in Θ. We must show that there is a time after which every event $msuspect(P)$ generated by correct process $p \in \Theta$ are such that $\Psi \subseteq P$. By Strong Completeness, there is a time t_1 after which every correct process $p \in \Theta$ will permanently suspect the processes in Ψ. It follows that every message $\langle M, A, F\rangle$ sent by each $p \in \Theta$ after t_1 is such that $\Psi \subseteq F[q]$. The proof is by induction on $|\Theta|$. If $|\Theta| = 1$, after t_1, the set $fd_p[p]$ (and $reachable_p$) is permanently equal to $\{p\}$; thus, the claim trivially holds. If $|\Theta| = n$, let Θ' denote the set $\{q | q \in \Theta \wedge \exists \bar{t}, \forall t > \bar{t} : \Psi \subseteq fd_p[q](t)\}$. If $\Theta = \Theta'$, then there is a time after which $reachable_p \cap \Psi = \emptyset$. If, on the other end, $\Theta' \subset \Theta$, none of the messages sent by processes in $\Theta - \Theta'$ after t_1 is received by processes in Θ'; by Eventual Symmetry, there is a time t_2 after which Θ' and $\Psi' = \Psi \cup (\Theta - \Theta')$ are definitely unreachable. Since $|\Theta'| < |\Theta|$, the proof is concluded by induction.
2. Let q be a process always reachable from p after time t_0; we must show that there is a time after which, given a process r, for each event $msuspect(P)$ generated by r, $q \in P$ if and only if $p \in P$. Let Θ denote the set of processes such that there is a time t_1 after which every message $\langle M, A, F\rangle$ sent by a process in Θ is such that $q \notin F[p]$. By Eventual Strong Accuracy, at least p and q belong to Θ. We claim that none of the processes not in Θ receives messages from processes in Θ sent after t_1. By contradiction, suppose a process $r \notin \Theta$ receives a message $\langle M, A, F\rangle$ sent by a process in Θ after t_1. Thus, $q \notin F[p]$ and r removes q from $fd_r[p]$ after t_1. Since q cannot be inserted again in $fd_r[p]$, there is a time after which all messages $\langle M, A, F\rangle$ sent by r are such that $q \notin F[p]$, thus $r \in \Theta$, a contradiction. By the Eventual Symmetry and Fair Channels properties, Θ and $\Pi - \Theta$ are permanently unreachable after t_1. If $r \notin \Theta$, from 1. it follows that there is a time after which every event $msuspect(P)$ generated by r is such that $p, q \in P$. If $r \in \Theta$, there are two possibilities. If r generates an event $msuspect(P)$ for q after t_1 due to a change in $reachable_r$, P must contain p also (by $reachable_r$ construction). If r generates an $msuspect$ event for q after t_1 due to the receipt of a (SUSPECT, –, G) tuple, it is easy to see that $p, q \in G$, so r generates an $msuspect$ event for both.
3. We must show that a process generates an $mrecv$ event for a message m at most once and only if some process q actually m-sent it earlier to a set of processes containing p. Before generating an $mrecv$ event for a message m m-sent by a process q, p checks that m has not been delivered yet, by verifying that the sequence number of m immediately follows the value stored in $ack_p[p][q]$; since, after the first $mrecv$ event, $ack_p[p][q]$ is set equal to the sequence number of m, p generates a $mrecv$ event for m at most once. As regards the second part, p generates an event $mrecv(m, q)$ at time t_0 only after executing an event $msend(m, G)$ such that $p \in G$ or after the receipt of a message $\langle M, A, F\rangle$ such that $(\text{MSEND}, m, -) \in M[q][p]$. The first case is trivial. In the second case, there is a time $t_1 < t_0$ at which $msg_q[q][p]$ contained $(\text{MSEND}, m, -)$; this implies that q has executed an event $msend(m, G)$ at time $t_2 < t_1$ such that $p \in G$.
4. We must show that messages from the same sender are m-received in FIFO order. Let m be a message m-sent by a process q. When p has m-received m, it has set $ack_p[p][q]$ equal to the sequence number associated with it. By construction, all the messages from q m-received by p after m have a sequence number greater than the number associated with m. Hence, they have been m-sent after m.
5. Let p be a correct process that m-sends at time t_0 a message m to a set of processes containing q. We must show that either (a) q will eventually generate a $mrecv$ event for m or (b) p will eventually generate a $msuspect$ event for q. By contradiction, suppose the claim is false. This implies that m will never be removed from $msg_p[p][q]$ after t_0. Let Θ denote the set of processes that never receive a message $\langle M, A, F\rangle$ such that $(\text{MSEND}, m, n) \in M[p][q]$. By definition, all processes not in Θ receive at least one message $\langle M, A, F\rangle$ such that $(\text{MSEND}, m, n) \in M[p][q]$. Since q never performs an $mrecv$ event for m, q never generates an acknowldgement for m. Thus, there is a time $t_1 > t_0$ after which all messages $\langle M, A, F\rangle$ sent by processes not in Θ that $\langle \text{MSEND}, m, n\rangle \in M[p][q]$ holds. By the Fair Channels property, processes in Θ are permanently unreachable from processes not in Θ after t_1. By definition, p is not in Θ. There are two possibilities: If $q \in \Theta$, by 1. it follows that there is a time after which every event $msuspect(P)$ generated by p is such that $q \in P$, a contradiction. Thus, suppose $q \notin \Theta$. By hypothesis, p never m-suspects q after having m-sent m. Thus, p never inserts a $(\text{SUSPECT}, n', P)$ tuple in $msg_p[p][q]$ such that $n' > n$. The proof continues by induction on n. If $n = 1$, q must m-receive m (a contradiction) since q cannot receive a tuple SUSPECT with a value greater than n. Then,

suppose $n > 1$. This implies that q cannot m-receive the message since its variable $ack_q[q][p]$ is blocked on a value $n' < n - 1$. There are two possibilities: The value $n' + 1$ is associated either with a regular message m' or with a tuple SUSPECT. In the first case, by inductive hypothesis, q will m-receive m' or p will m-suspect q after having m-sent m' (and q will m-receive a SUSPECT tuple with a value greater than $n' + 1$); this implies that the value of $ack_q[q][p]$ will eventually increase, a contradiction. In the second case, q will m-receive a SUSPECT tuple containing $n' + 1$ and the proof terminates as in the previous case.

6. We must show that no process generates $msuspect$ events for itself. There are two cases in which a process p generates a $msuspect$ event: (a) when a change occurs in the $reachable_p$ set in which case p generates an $msuspect$ event for the set $\Pi - reachable$, which cannot contain p; or (b) after receiving a message $\langle M, A, F\rangle$ where $(\text{SUSPECT}, -, G) \in M[q][p]$ for some process q. Since q has inserted the SUSPECT tuple in $msg[q][p]$ immediately after having excluded p from $reachable_q$ and G is equal to $reachable_q$, G does not contain p. So, p never generates an $msuspect$ event for itself.
7. Let p be a process that never generates an $msuspect$ event for q after time t_0. We must show that there is a time after which q never generates an $msuspect$ event for p. By contradiction, suppose this is false. By the algorithm, there is a time $t_1 > t_0$ at which $msg_q[q][p]$ contains a $(\text{SUSPECT}, n, -)$ tuple not yet received by p. Let Θ be the set of processes r such that there is a time at which $(\text{SUSPECT}, n, -) \in msg_r[p][q]$. If $p \in \Theta$, then p eventually generates an $msuspect$ event for q after t_0 as a result of the receipt of $(\text{SUSPECT}, n, -)$. If, on the other end, $p \notin \Theta$, since the sets Θ and $\Pi - \Theta$ are permanently unreachable and $q \in \Theta$, from 1. it follows that p generates an $msuspect$ event for q after time t_0. □

APPENDIX B

VIEW MANAGEMENT LAYER: DETAILED DESCRIPTION

In this appendix, we present a detailed description of VML illustrated in Figs. 2, 3, 4, and 5.

B.1 Variables and Messages

The local state of each process p is defined by the global variables $view$, $cview$, $reachable$, $version$, $agreed$, $estimate$, $symset$, $stable$, and $ctbl$. Variable $view$ is composed of the fields $view.id$ and $view.comp$ that represent, respectively, the identifier and the composition of the last view passed up to the application. Variable $cview$ has the same fields as $view$, but contains the complete view that has been m-received with the last VIEW message. When the current view of p is a complete view, $view$ and $cview$ are equal. $reachable$ corresponds to the set of processes perceived to be reachable (i.e., not m-suspected by MSL). During idle phase, $reachable$ and $view.comp$ coincide. A new agreement phase is started whenever $reachable$ changes due to a $msuspect$ event or when $reachable$ is different from the composition of the last installed view. $version$ is an array indexed by Π; for each $q \in \Pi$, $version[q]$ contains the last version number of q known by p. $version[p]$ is the current version number of p. Version numbers are generated by processes whenever they enter a new agreement phase. When a process enters the ee-phase, it creates a new array of version numbers called $agreed$. While $version$ can continually change during an ee-phase, $agreed$ is fixed and is used to identify the messages related to this particular ee-phase. $estimate$ is a set of processes and represents the proposal for the composition of the next view. $symset$ is an array indexed by Π; for each $q \in \Pi$, $symset[q]$ contains the last value stored in variable $reachable$ such that $q \notin reachable$. This information, communicated by p to q through SYNCHRONIZE messages, is used by the algorithm to satisfy Property GM1. $stable$ is a Boolean variable indicating whether the last installed view corresponds to the approximation of reachability supplied by the MSL or the process has to enter agreement phase again. $ctbl$ is the *coordinator table* and contains all information needed by a process when it assumes the role of coordinator. In particular, $ctbl$ is an array of records indexed by Π, where each record contains the entries $cview$, $agreed$, and $estimate$. For each q, $ctbl[q]$ represent p's perception of the value of those variables at process q. Finally, there are three variables that have only local scope: $event$ is used with the **wait-for** construct and contains the last event that occurred at p; $synchronized$ is used in the s-phase and contains the set of processes from which p has m-received an answer to its synchronize request; $installed$ is a Boolean variable used in procedure $EstimateExchangePhase()$, whose value becomes true when the agreement protocol can terminate.

All messages that are m-sent contain a tag indicating the message type plus other fields relevant for that type. Five message types are used by the algorithm. $\langle \text{SYNCHRONIZE}, V_p, V_q, P\rangle$ messages are used during the s-phase. Fields V_p and V_q represent the version numbers of the sender and the destination as known by the sender at the time of m-sending, and P represents the last $symset$ associated with the destination process. When a process enters the s-phase, it m-sends a SYNCHRONIZE message to each reachable process that responds to another SYNCHRONIZE message. $\langle \text{SYMMETRY}, V, P\rangle$ messages are used in both s-phase and ee-phase to handle situations where approximations of reachability obtained by the MSL are temporarily asymmetrical. Here, V is a version number array, while P is the approximation of the set of reachable processes known by p at the time of m-sending. $\langle \text{ESTIMATE}, V, P\rangle$, $\langle \text{PROPOSE}, S\rangle$, and $\langle \text{VIEW}, w, C\rangle$ messages are used during the ee-phase. A ESTIMATE message is m-sent to processes belonging to the current estimate whenever this estimate changes. Once again, V is a version number array, while P is the set of processes in the current estimate of the sender. PROPOSE messages are m-sent by the processes to the coordinator each time the

next view estimate changes; the field S contains the status of the sender that corresponds to a $ctbl$ entry. Finally, VIEW messages are m-sent by the coordinator to processes when agreement is reached; field w is the new complete view identifier, while C is the value of the coordinator table when agreement has been reached and contains the composition of the view and other information used to construct partial views (if necessary). We say that a process *accepts* an m-received message if it modifies its status according to the contents of the message.

B.2 Algorithm Description

Instead of giving an exhaustive description of the entire algorithm, we illustrate some of its peculiarities in order to simplify the understanding of the overall structure. Fig. 2 contains the main body of the algorithm. During the first part of procedure $ViewManagement()$, process p initializes some variables and installs a view containing only itself. In this way, every process can independently create its initial view without having to reach agreement with any other process. This view will last until the first $msuspect$ event occurs. Variables $view$ and $cview$ are updated in order to simulate the m-receipt of a VIEW message generated by p itself. $UniqueID()$ is a predefined function used to create new identifiers. After initialization, the process remains idle until an event occurs causing it to enter agreement phase.

There are two conditions under which a process enters agreement phase: The execution of an $msuspect(P)$ event or the m-receipt of a SYNCHRONIZE message. The second case is straightforward: p enters agreement phase if the message is not obsolete and the sender is believed reachable (remember that a process may m-receive messages from unreachable processes due to fact that MSL guarantees only eventual, and not perpetual, symmetry between processes). As for the first case, the code associated with this event must be described carefully since it is related both to the agreement protocol termination and to View Accuracy. Furthermore, a similar code is repeated in other parts of the algorithm. First of all, p updates the $symset$ array entries of all processes that have become reachable since the previous $msuspect$ event and m-sends a SYMMETRY message to them. The aim of this code is to reestablish symmetry on view estimates whenever the approximation obtained by the MSL is temporarily asymmetrical. To see this point, consider the following scenario. Let q be a process engaged in agreement phase waiting for agreement to be reached with p. Suppose p temporarily m-suspects q before the agreement can be reached. As we noted in the introduction, p removes q from its view estimate and the exclusion is permanent during this agreement phase. Note that p may reinsert q into the view estimate in the next view agreement. Process q, however, cannot participate in the new agreement phase of p until it has terminated its previous agreement phase in order to prevent propagation of obsolete exclusions. Thus, q cannot install a new view since it waits forever for the participation of p in its current agreement phase. The SYMMETRY message forces q to remove p from its view estimate and allows q to reach an agreement on a new view. Furthermore, note that each SYMMETRY message also carries a set of processes corresponding to the value of variable $reachable$ of the sender. This set is necessary to satisfy Property GM1: When a process m-receives a SYMMETRY message, it excludes from its view estimate all processes believed reachable by the sender. In this manner, if p m-receives a SYMMETRY message from q and q and r are permanently reachable, p removes r as well and it will not m-send an ESTIMATE message to r inviting it to remove q. The value $reachable$ is stored in array $symset$ to be used in the same way with SYNCHRONIZE messages. After these steps, p updates variable $reachable$ and calls procedure $AgreementPhaset()$.

Procedure $AgreementPhase()$, illustrated in Fig. 3, implements the agreement phase. After having initialized the next view estimate and having generated a new version number, process p calls procedure $SynchronizationPhase()$ and then $EstimateExchangePhase()$. These actions are repeated until the current view is "stable," meaning that the view composition coincides with the set of reachable processes and no new agreements have been initiated (see procedure $InstallView()$ for details).

As the first step in procedure $SynchronizationPhase()$, process p initializes variable $synchronized$ (which contains processes that know p's new version number) to be equal to the singleton set $\{p\}$ and m-sends SYNCHRONIZE messages announcing its new version number. Note that if p has entered the s-phase due to a SYNCHRONIZE message m from a process q, the message m-sent by p also acts as a reply to m. S-phase lasts until all processes in the view estimate have replied to the SYNCHRONIZE message of p or when p m-receives a message from a process in its view estimate that has already entered the ee-phase. In the **while** loop, this condition is encoded as $synchronized \not\subseteq estimate$. To guarantee the termination of this phase, every time p m-suspects a process q or m-receives a SYMMETRY message from q, p removes q from its view estimate. This exclusion cannot be revoked during the current view. On the contrary, when p m-receives a SYNCHRONIZE message containing its last version number from a process q, it adds q to $synchronized$. In this manner, $synchronized$ will eventually contain $estimate$. Another method for p to enter ee-phase is to m-receive a message $\langle \text{ESTIMATE}, V, P\rangle$ from a process q that knows p's current version number and has not been removed from p's view estimate. If so, p modifies $synchronized$ and $estimate$ in order to guarantee the exit from the s-phase. In any case, during the s-phase, p constructs the version number array $agreed$ that will be used in the ee-phase to discard obsolete messages.

Fig. 5 illustrates the three procedures used during the ee-phase. Procedure $SendEstimate(P)$ is used to modify the view estimate and to inform the other processes of the change. At the same time, during this procedure, p m-sends a PROPOSE message carrying its current status to a coordinator selected from its view estimate through function $Min()$. Function $CheckAgreement(C)$ is used when a process m-receives PROPOSE messages. It verifies whether the proposals stored in the coordinator table are in agreement by checking if all of the view estimates are equal and if all processes know the same version number arrays (restricted to processes in the view estimate itself). Finally, procedure $InstallView$ is used to install a new view. First of all, p forwards a VIEW message containing the new

```
1   mcast(m):
2       dlvr(m)
3       buffer ← buffer ∪ {m}
4       msend(⟨MULTICAST, view.id, m⟩, view.comp − {p})
5
6   mrecv(⟨MULTICAST, w, m⟩, q):
7       if (view.id = w) then
8           dlvr(m)
9           buffer ← buffer ∪ {m}
10      fi
```

Fig. 8. Reliable multicast service extension to idle phase and s-phase for process p.

complete view identifier w and the coordinator table C to all processes that have reached the agreement. This operation is necessary since the coordinator may crash before m-sending the VIEW message to all its recipients. Then, p checks whether a partial view is needed or not. Suppose there exists a process q whose last view v contains a process r, but the last view installed by r is different from v. If p installed the complete view, this would violate Property GM3. For this reason, the complete view is split into a set of partial views. Each partial view is composed of the set of processes that have m-received the same previous complete view and is identified through a pair (w, v) composed of the identifiers of the new complete view and the previous view. Note that all processes in each partial view compute, starting from C, the same identifier and composition. Otherwise, if there is no need of partial views, p will install the complete view, identified by the pair $(w, \perp)$, where $\perp$ means that w is a complete view.

Finally, Fig. 4 contains the main body of the ee-phase. During the ee-phase initialization (procedure $Initialize\ EstimatePhase()$), p calls procedure $SendEstimate()$ in order to guarantee the m-sending of at least one ESTIMATE and one PROPOSE message. Then, p enters a loop from which it will exit only when $installed$ becomes true. During the ee-phase, processes exchange ESTIMATE messages to promote the reaching of an agreement. A message $\langle \text{ESTIMATE}, V, P \rangle$ is accepted if, for each r in the intersection between the estimate of p and the estimate of q, the version number $agreed[r]$ known by p is equal to $V[r]$ known by q. This check is needed to guarantee that p never accepts an ESTIMATE message containing an estimate generated before the start of the previous s-phase that could cause exclusion of reachable processes. If the message is accepted, p calls procedure $SendEstimate()$, removing processes not in P. Note that, in case $p \notin P$, the ESTIMATE message is interpreted as a SYMMETRY message. Apart from ESTIMATE and SYMMETRY messages and direct $msuspect()$ events, during ee-phase a process p may exclude a process from its view estimate also through SYNCHRONIZE messages. When a process m-receives a message $\langle \text{SYNCHRONIZE}, V_p, V_q, P \rangle$ from a process q such that V_q is greater than $agreed[q]$, p is informed that q has completed the agreement phase identified by $agreed[q]$ and has installed a new view. There are two possibilities: p does not belong to the view installed by q, thus p can exclude q from its view estimate, or p belongs to the view installed by q, but the VIEW message m-sent by q to p has been lost (by the FIFO order condition of MSL). This is another good reason for p to exclude q from its view estimate. In order to not violate View Accuracy, p removes all processes contained in P.

Appendix C

Reliable Multicast Service: Implementation

In this appendix, we show how to extend our solution for PGMS to provide the reliable multicast service specified in Section 6. Both the idle and the agreement phases of the algorithm have to be modified: The idle phase must handle $mcast()$ events, while the agreement function must verify

```
1   mcast(m):
2       suspended ← suspended ∪ {(⊥, m)}
3
4   mrecv(⟨MULTICAST, w, m⟩, q):
5       suspended ← suspended ∪ {(w, m)}
6
7   mrecv(⟨DELIVERED, w, B⟩, q):
8       if (view.id = w) and (q ∈ estimate) then
9           foreach(m ∈ B − buffer) do
10              dlvr(m)
11          msend(⟨DELIVERED, view.id, B − buffer⟩)
12          buffer ← buffer ∪ B
13          msend(⟨PROPOSE, (cview, agreed, estimate, buffer)⟩, Min(estimate))
14      fi
```

Fig. 9. Reliable multicast service extension to ee-phase for process p: part (a).

```
1   procedure InitializeEstimatePhase()
2       SendEstimate(∅)
3       msend(⟨DELIVERED, view.id, buffer⟩)
4
5   procedure SendEstimate(P)
6       estimate ← estimate − P
7       msend(⟨ESTIMATE, agreed, estimate⟩, reachable − {p})
8       msend(⟨PROPOSE, (cview, agreed, estimate, buffer)⟩, Min(estimate))
9
10  function CheckAgreement(C)
11      return   (∀q ∈ C[p].estimate : C[p].estimate = C[q].estimate ∧ C[p].buffer = C[q].buffer)
12               and (∀q, r ∈ C[p].estimate : C[p].agreed[r] = C[q].agreed[r])
13
14  procedure InstallView(w, C)
15      msend(⟨VIEW, w, C⟩, C[p].estimate − {p})
16      if (∃q, r ∈ C[p].estimate : q ∈ C[r].cview.comp ∧ C[q].cview.id ≠ C[r].cview.id) then
17          view ← ( (w, view.id), {r | r ∈ C[p].estimate ∧ C[r].cview.id = cview.id} )
18      else
19          view ← ((w, ⊥), C[p].estimate)
20      generate vchg(view)
21      cview ← (w, C[p].estimate)
22      stable ← (view.comp = reachable) and (∀q, r ∈ C[p].estimate : C[p].agreed[r] = agreed[r])
23      buffer ← {m | (⊥, m) ∈ suspended ∨ (view.id, m) ∈ suspended}
24      foreach m ∈ buffer do
25          dlvr(m)
26      foreach (⊥, m) ∈ suspended do
27          msend(⟨MULTICAST, view.id, m⟩, view.comp − {p})
28      suspended = ∅
```

Fig. 10. Reliable multicast service extension to ee-phase for process p: part (b).

that Property RM1 is satisfied before declaring that agreement has been reached. Figs. 8, 9, and 10 contain the changes to the original algorithm of Section 5.1. The code of Fig. 8 contains two new event handling procedures that have to be inserted in the case statement of both the idle phase (Fig. 2) and the s-phase (Fig. 3). The code of Fig. 9 contains three new event handling procedures that have to be added to the case statement of the ee-phase (Fig. 4). Finally, the code in Fig. 10 substitutes the corresponding functions of Fig. 5 used during ee-phase.

We introduce two new global variables that are used to implement view synchrony. The first, called *buffer*, contains the set of messages delivered by the process during its current view and is used to allow the coordinator to verify agreement on messages. The second is called *suspended* and contains a set of pairs (v, m), where m is a message whose delivery cannot be performed in the current view, but has to be postponed to view v in order to satisfy Property RM1. If v is equal to the special value $\perp$, then m has to be postponed to the next view. Both sets are initialized to be empty. Furthermore, two new message types are needed. A $\langle \text{MULTICAST}, v, m \rangle$ message is used to inform the other processes that m has to be delivered during view v; a $\langle \text{DELIVERED}, B, v \rangle$ is used to inform the other processes that during v, the sender has delivered the set of messages contained in B. MULTICAST messages are accepted only during idle phase or s-phase, while DELIVERED messages are accepted only during the ee-phase. DELIVERED messages are needed to allow processes to reach agreement on the set of messages to be delivered during the last installed view.

Some changes are also needed for existing variables and messages. Variable *ctbl*, the coordinator table describing the current status of each process participating in an agreement, is extended to contain a field called *buffer* to be used for agreement on the set of delivered messages. For the same reason, PROPOSE messages must contain the *buffer* field as well.

Whenever a process p wants to multicast a message m, it invokes the primitive $mcast(m)$. The actions of the algorithm are different depending on the current phase. Let v be the current view of process p. During the idle and s-phases (Fig. 8), p m-sends a $\langle \text{MULTICAST}, v, m \rangle$ message to all processes in $\overline{v}$ (excluding p itself). Then, p locally delivers the message and adds it to *buffer*. If the multicast request occurs in the ee-phase that was started during view w (Fig. 9), the pair $(\perp, m)$ is added to *suspended* where the place holder $\perp$ means that m has to be multicast at the beginning of the next view. We use variable *suspended* to avoid a process delivering new messages during the ee-phase after having m-sent a PROPOSE message that can lead to an agreement. Otherwise, let v be the current view of p. Suppose p delivers a message m during the ee-phase, then m-receives a VIEW message from a process $q \in \overline{v}$ that has installed the next view w but has not delivered m. If p installs the view, then p and q will not have delivered the same set of messages during v. Otherwise, by GM3, q must

install a new view excluding p. Obviously, this may violate GM1 if p and q are permanently reachable.

The behavior of the algorithm depends on the current phase even when a process m-receives a $\langle \text{MULTICAST}, w, m \rangle$ message. Let v be the current view of p. If the message is m-received during the idle phase or the s-phase (Fig. 8), p delivers the message and adds it to *buffer* if and only if v is equal to w. If the message is m-received during the ee-phase (Fig. 9), the pair (w, m) is inserted in $suspended$ because the MULTICAST message may have been generated by a process q that has already installed the next view.

Finally, a few changes are necessary in other parts of the ee-phase. During the ee-phase initialization (procedure $InitializeEstimatePhase()$), each process m-sends a $\langle \text{DELIVERED}, B, v \rangle$ message to processes in its view estimate, where B is equal to *buffer*. When a process m-receives a DELIVERED message from a process not excluded from its view estimate (Fig. 9), it delivers all undelivered messages contained in B and adds them to *buffer*. Moreover, it m-sends a new $\langle \text{DELIVERED}, B', v \rangle$ message, where B' contains the subset of messages not yet delivered before the receiving of B. This additional message is necessary. p could accept DELIVERED messages from processes included in its current view estimate, but excluded from the estimate of other processes participating in the agreement. Finally, process p m-sends a PROPOSE message to the current coordinator. Function $CheckAgreement()$ (Fig. 10) will return true only if all processes that will survive from a view in the next one (partial or complete) have delivered the same set of messages. Finally, after having installed the new view w, function $InstallView()$ (Fig. 10) must deliver all messages whose delivery in the previous view had to be postponed.

Property RM1 is guaranteed by the fact that, during the ee-phase, each process can deliver only messages previously delivered by processes in the current view estimate. Moreover, agreement on a new view may be observed only when the proposals m-sent by processes surviving from a view to the same successor view contain the same set of messages. Property RM2 is guaranteed by the fact that each message is associated with the view in which it must be delivered. Property RM3 is guaranteed by the creation of partial views. Property RM4 is straightforward, while Property RM5 follows from the liveness condition of MSL and from GM3.

Once again, the aim of the algorithm we have presented is to show the implementability of our specifications. Many possible optimizations have been neglected for the sake of simplicity. For example, acknowledgment information can be piggybacked in $mcast$ messages to decrease the size of *buffer*.

APPENDIX D

PARTITIONABLE GROUP MEMBERSHIP ALGORITHM: CORRECTNESS PROOF

In this appendix, we illustrate the correctness of the PGMS algorithm presented in Section 5. As in Appendix A, variable names are indexed with process names. Furthermore, we refer to the s-phase (ee-phase) *of a view* v to denote the s-phase (ee-phase, respectively) started *during* view v. Finally, we say that a message $\langle \text{VIEW}, w, C \rangle$ *contains* a view v if v is either equal to w or is a partial view obtained from w and C.

First of all, we must prove that any correct process that invokes the agreement protocol will eventually install a new view. This termination property is fundamental since it will be used in the proofs of Properties GM1, GM2, and GM3. The proof is divided into two parts: 1) If a correct process enters the s-phase of a view v, it will eventually enter the ee-phase of v itself; 2) if a correct process enters ee-phase of a view v, it will eventually install a view after v. Since the second part is needed in the proof of the first one, we first prove 2) and then 1).

Lemma D.1. *The number of PROPOSE messages m-sent by a process during a view is bound.*

Proof. Apart from the PROPOSE message m-sent by p immediately after entering the ee-phase of a view, a process m-sends a PROPOSE message only when it modifies its variable $estimate$. By construction, no process may be added to $estimate$ during a view after its initialization. Since the starting cardinality is bound, the number of PROPOSE messages m-sent by a process during a view is bound as well. □

Lemma D.2. *If a correct process p enters the ee-phase of a view v, then p will eventually install a new view after v.*

Proof. By contradiction, suppose p installs a bound number of views and v is the last view installed by p. Let V_p denote the value of $agreed_p$ when p entered the ee-phase of v. By Lemma D.1, the number of different PROPOSE messages m-sent by p during v is bound. Let $\langle \text{PROPOSE}, s_p \rangle$ be the last PROPOSE message m-sent by p (such message exists since p m-sends at least one PROPOSE message at the beginning of the ee-phase). Let P_p denote the set $s_p.estimate$. By Property A.1.6 and, because only m-suspected processes may be removed from variable $estimate$, P_p is not empty. Because processes in P_p cannot be m-suspected from p during v, by Property A.1.1, every process in P_p is correct. Moreover, every message m-sent by p during v to a process in P_p will eventually be m-received.

For each process $r \in P_p$, the version number $V_p[r]$ is generated at the beginning of the s-phase of a view v_r. First of all, we must prove the following:

Claim. *Each process $r \in P_p$ will eventually enter ee-phase during v_r.*

By contradiction, suppose the claim is false. Thus, r is blocked in the s-phase of v_r. At the beginning of ee-phase of view v, p has m-sent to r a message $\langle \text{ESTIMATE}, V_p, - \rangle$. When r m-receives it, there are two possibilities:

- *If r has excluded p from its view estimate, it m-sends a SYMMETRY message to p containing $V_p[p]$; by Properties A.1.7 and A.1.5, p will eventually m-receive such a message and exclude r from its view estimate, a contradiction.*
- *Otherwise, r accepts the ESTIMATE message and enters ee-phase. This concludes the claim.*

The next step consists of showing that all processes in P_p are blocked in the ee-phase:

Claim. *Each process $r \in P_p$ is blocked in the ee-phase of its view v_r.*

By contradiction, suppose the claim is false. Thus, r installs another view after v_r. Note that p cannot m-receive a SYNCHRONIZE or a SYMMETRY message from r containing a version number array V such that $V[r] > V_p[r]$. Otherwise, p would exclude r from its view estimate, a contradiction. By Property A.1.7, there is a time t_1 after which r stops m-suspecting p. There are two possibilities:

- *If r installs an unbound number of views, p will m-receive an unbound number of different SYNCHRONIZE messages containing increasing version numbers for r, a contradiction.*
- *If r installs a bound number of views, let v'_r be the last view installed by r. First of all, we must show that r will enter agreement phase during v'_r. If $\overline{v'_r}$ does not contain p, r enters agreement phase immediately after having installed v'_r or when it stops m-suspecting p. If $\overline{v'_r}$ contains p, r forwards the VIEW message containing v'_r to p. There are two possibilities:*
  - *If this message is lost, by Property A.1.5, r will m-suspect p during v'_r and enter the agreement phase;*
  - *if p m-receives the message, by hypothesis, p discards it, but this implies that the agreement on v'_r has been reached with a PROPOSE message of p m-sent before the installation of v. Thus, r will eventually enter agreement phase.*

  At the beginning of the new agreement phase, r generates a new version number n_r greater than $V_p[r]$. If this happens before t_1, p will m-receive from r a SYMMETRY message containing n_r; otherwise, if this happens after t_1, p will m-receive from r a SYNCHRONIZE message containing n_r. In both cases, we have obtained a contradiction that concludes the claim.

Given that all processes in P_p are blocked in the ee-phase, by Lemma D.1, the number of different PROPOSE messages m-sent by each process $r \in P_p$ during v_r is bound. Let $\langle \text{PROPOSE}, s_r \rangle$ be the last PROPOSE message m-sent by r and let P_r denote the set $s_r.estimate$:

Claim. *For each process $r \in P_p$, p belongs to P_r.*

By contradiction, suppose the claim is false. By Property A.1.7, there is a time after which r stops m-suspecting p. Then, p will m-receive a message $\langle \text{ESTIMATE}, V, P \rangle$ such that $V[r] = V_p[r]$ and $p \notin P$. Thus, p will remove r from its view estimate, a contradiction.

Note that the claims we have developed so far can be applied to each of the processes in P_p. Given a process r, we can show that:

- Each process $s \in P_r$ installs a bound number of views and is blocked in the ee-phase of its last view v_s
- r knows the last version number of each process in P_r.

Thus, all processes in P_p know the same version number for each of the processes in P_p. By Property A.1.7, each process $r \in P_p$ will accept at least one message $\langle \text{ESTIMATE}, -, P_p \rangle$ from p. p will m-receive and accept at least one message $\langle \text{ESTIMATE}, -, P_r \rangle$ as well. Since we have supposed for each process $r \in P_p$ that P_r is the last view estimate, then, for each process $r \in P_p$, we have that $P_r \subseteq P_p$ and $P_p \subseteq P_r$. Thus, all values P_r are equal to P_p.

This implies that all processes in P_p maintain the same view estimate P_p and know the same agreed version number array (restricted to the processes in P_p). By construction, they m-send their last PROPOSE messages to the same coordinator. Since no process in P_p can m-suspect another process in P_p, the coordinator will m-receive all these messages and observe an agreement. Thus, the coordinator will m-send a VIEW message to p, then m-receives it and installs a new view, a contradiction that concludes the proof. □

Lemma D.3. *If a correct process p enters the s-phase of a view v, then p will eventually enter the ee-phase of v.*

Proof. By contradiction, suppose p never enters the ee-phase of v. Under this assumption, during the s-phase, no process can be added to $estimate$ or removed from $synchronized$ (note that processes may be added to $estimate$ or removed from $synchronized$ during the s-phase, but only just before entering the ee-phase of v). Thus, there is a time t_1 after which p never modifies its variables $estimate_p$ and $synchronized_p$. Let n_p denote the final value of $version_p[p]$ (recall that a process generates new version numbers only when it enters a new agreement phase) and let q be a process contained in $estimate_p(t_1)$, but not in $synchronized_p(t_1)$ (such a process must exist, otherwise p will enter the ee-phase of v). By hypothesis, p cannot m-suspect q after having entered the s-phase of v. By Property A.1.1, q is correct. By Property A.1.5, q will eventually m-receive the SYNCHRONIZE message containing n_p that was m-sent by p at the beginning of the s-phase of v. And, it will store n_p in $version_q[p]$ at time t_2. If q m-suspects p after t_2, by Properties A.1.7 and A.1.5, p will eventually m-receive a message $\langle \text{SYMMETRY}, V, - \rangle$ such that $V[p] = n_p$ and it will remove q from $estimate_p$, a contradiction. Thus, suppose q never m-suspects p after t_2. There are two possibilities:

- If q is either in the idle phase or in the s-phase at time t_2, by construction and by Property A.1.5, p will eventually m-receive a message $\langle \text{SYNCHRONIZE}, n_p, -, - \rangle$. Thus, p will insert q in $synchronized_p$, a contradiction.
- Otherwise, suppose q is in the ee-phase at time t_2. By Lemma D.3, q will eventually install a new view w after t_2 due to the m-receipt of a message $\langle \text{VIEW}, w, C \rangle$. There are two possibilities:
  - If $\overline{w}$ does not contain p, the new view does not correspond to the current estimate of the reachability set of q (p is not m-suspected). Thus, after the installation of w, variable $stable$ is set to false.

- If $\overline{w}$ contains p, $C[q].version[p]$ is different from $version_q[p] = n_p$ (since p has never m-sent a PROPOSE message containing n_p). Thus, after the installation of w, variable $stable$ is set to false.

In both cases, q enters the agreement phase again and m-sends a SYNCHRONIZE message containing n_p to p. By Property A.1.5, p will m-receive it and add q to $synchronized_p$, a contradiction. □

Corollary D.1. *If a correct process p enters the agreement phase during a view v, then it will eventually install a new view after v.*

Proof. From Lemmas D.2 and D.3. □

The next property to prove is View Accuracy. In the following, we say that a process r excludes a process s from its view estimate *indirectly* if the exclusion follows the m-receipt of an $\langle \text{ESTIMATE}, -, P\rangle$ message such that $r \in P$ and $s \notin P$. Otherwise, we say that r excludes a process s from its view estimate *directly* (for example, due to an m-suspect or a SYMMETRY message).

Lemma D.4. *Let v be a view installed by a process p and let t be the time at which p entered the s-phase of v. Let q be a process from which p accepts a message $\langle \text{ESTIMATE}, -, P\rangle$ such that $p \in P$, m-sent by q during a view v_q. Then, q has entered the ee-phase of v_q after t.*

Proof. At the beginning of the s-phase of v, p m-sends a SYNCHRONIZE message containing its current version number n_p. By Lemma D.3, p will eventually enter the ee-phase of v. There are two possibilities:

- p enters the ee-phase of v after having m-received a SYNCHRONIZE message containing n_p from each process not m-suspected after the beginning of the s-phase of v. By construction, all processes in $estimate_p$ were in the s-phase when they m-sent the SYNCHRONIZE reply containing n_p. Thus, they enter the ee-phase of their view after t.
- p enters ee-phase of v after having m-received an ESTIMATE message containing n_p m-sent by a process q_1 that has entered the ee-phase after storing n_p in its array $version_{q_1}$. In this case, we again have two possibilities: q_1 has entered the ee-phase either after having m-received a SYNCHRONIZE message from each process not m-suspected during the s-phase or after having m-received an ESTIMATE message m-sent by a process q_2 that has entered the ee-phase. By iterating the reasoning, we obtain a finite chain $p, q_1, \cdots, q_n$ such that each process has entered the ee-phase after having m-received an ESTIMATE message from the following one, apart from q_n which has m-received a SYNCHRONIZE message from each process not m-suspected during the s-phase. Let t_i denote the time at which q_i m-sends the ESTIMATE message. By construction, we have that $estimate_{q_i}(t_i)$ is contained in $estimate_{q_{i+1}}(t_{i+1})$, for each $i = 1 \ldots n-1$. Thus, $p \in estimate_{q_i}(t_i)$, for each $i = 1 \ldots n$. Moreover, we have that $version_{q_i}[p](t_i)$ is equal to $version_{q_{i+1}}[p](t_{i+1})$, for each $i = 1 \ldots n-1$. This implies that $version_{q_n}[p]$ is equal to n_p and that q_n has entered the ee-phase of its view after t. □

Theorem D.1 (View Accuracy). *The PGMS algorithm satisfies Property GM1.*

Proof. Let p be a correct process and let q be always reachable from p after time t_0. We must prove that there is a time after which the current view of p always contains q. Suppose p installs a bound number of views. In this case, the last view installed by p must contain q (since otherwise p would enter the agreement phase again and, by Corollary D.1, would install a new view, impossible by hypothesis). Thus, suppose p installs an unbound sequence of views.

Claim. *There is a time t_p after which no process distinct from p can directly exclude q from its view estimate without excluding p at the same time.*

By Property A.1.2, there is a time t_1 after which each process r m-suspects both or none of p and q. Since each message $\langle \text{SYMMETRY}, -, P\rangle$ m-sent by a process s at time t is such that P is equal to $reachable_s(t)$, it follows that there is a time t_2 after which all messages $\langle \text{SYMMETRY}, -, P\rangle$ m-received by a process r are such that P contains both or none of p and q. Moreover, we must prove that there is a time t_3 after which if a process r excludes q from its view estimate due to the m-receipt of a SYNCHRONIZE message, it excludes p as well. Recall that r may exclude q due the m-receipt of a message $\langle \text{SYNCHRONIZE}, V_p, V_s, -\rangle$ from a process s only if $agreed_r[s] < V_s$. By Property A.1.4 and by algorithm construction, this can happen only if r never m-receives an ESTIMATE or VIEW message m-sent by s before the SYNCHRONIZE message. Suppose there exists a process s from which r m-receives an unbound number of messages $\langle \text{SYNCHRONIZE}, V_p, V_s, P\rangle$ inviting r to exclude q from $estimate_r$ (otherwise the claim is trivially concluded). By Property A.1.5, s will m-suspect r an unbound number of times. Obviously, s will stop m-suspecting r an unbound number of times as well. Thus, s will modify its variable $symset_s[r]$ after time t_1. This implies that there is a time $t_3 c$ after which all messages $\langle \text{SYNCHRONIZE}, -, -, P\rangle$ m-sent by s to q are such that p, q belong both or none to P. This concludes the claim.

By Eventual Symmetry, there is a time after which p is always reachable from q. By repeating the reasoning of the previous claim, we obtain that there is a time t_q after which no process different from q can directly exclude p from its view estimate without excluding q as well.

The next step consists of showing that p and q eventually stop directly excluding each other:

Claim. *There exists a time $t_4 \geq t_p, t_q$ after which p cannot directly exclude q (and vice versa).*

By Properties A.1.6 and A.1.2, there is a time after which p cannot m-suspect q and q cannot m-suspect p. This implies that there is a time after which 1) p cannot m-receive and

accept a SYMMETRY or a SYNCHRONIZE message from a process distinct from q inviting p to exclude q (such messages cannot be m-sent to p), 2) p cannot m-receive a SYMMETRY message from q (since they stop m-suspecting each other), and 3) p cannot m-receive a SYNCHRONIZE message from q with a version number for q greater than $agreed_p[q]$ (by Property A.1.5, this is possible only if some messages from q to p have been lost, but we know that there is a time after which p is permanently reachable from q). In a symmetric way, we can prove that q cannot directly exclude p after t_4. This concludes the claim.

By hypothesis, p will install an unbound number of views after t_4. In order to conclude the proof, we must show that:

Claim. *There is a time t_{acc} after which all messages $\langle \text{VIEW}, -, C\rangle$ m-received and accepted by p are such that q belongs to $C[p].estimate$ (and vice versa).*

Let P_p be the set of processes that participate with p in the agreement of an unbound number of views. This implies that all processes in P_p are correct and invoke the agreement protocol an unbound number of times. By definition, there is a time t_5 after which p does not participate in the agreement of a view with a process not included in P_p. Consider a time t_6 at which all processes in P_p have installed at least one view after time $Max(t_4, t_5)$. Let v be a view installed by p after t_6. By Lemma D.4, all processes that participate with p in the agreement of the next view enter the ee-phase of their view after t_6. By construction, all these processes have entered the s-phase of their views after t_4. Thus, all messages $\langle \text{ESTIMATE}, -, P\rangle$ m-received by p during v are such that P contains both or none of p and q. This implies that p never removes q (directly or indirectly) from its view estimate during v. All $\langle \text{PROPOSE}, s\rangle$ messages m-sent by p during v are such that $q \in s.estimate$. Thus, the next message $\langle \text{VIEW}, w, C\rangle$ m-received by p is such that $q \in C[p].estimate$. This concludes the first part of the claim. In a symmetric way, we can prove that all messages $\langle \text{VIEW}, -, C\rangle$ m-received and accepted by q after t_{acc} are such that q belongs to $C[q].estimate$.

To conclude the proof, we must show that eventually each partial view installed by p contains q. Consider a message $\langle \text{VIEW}, v, C\rangle$ m-received and accepted by p. Let v_p and v_q denote the values $C[p].view.id$ and let $\overline{v_p}$ and $\overline{v_q}$ denote the values $C[p].cview.comp$ and $C[q].cview.comp$. Suppose the VIEW messages containing v_p and v_q have been m-received by p and q after t_{acc}, respectively. By contradiction, suppose v_p is different from v_q. Because p, q belong to both $\overline{v_p}$ and $\overline{v_q}$, when p m-receives $\langle \text{VIEW}, v, C\rangle$, it will install a partial view not containing q. p and q must have participated in the agreement of both v_p and v_q, but the corresponding VIEW messages must have been generated from two distinct coordinators, c_p and c_q, respectively. Suppose p has m-sent the PROPOSE message for the agreement on v_p before the PROPOSE message for v_q; by monotony of variable $estimate_p$ and by the use of function Min to select the coordinator, c_p cannot belong to $\overline{v_q}$. Thus, p has not m-received the $\langle \text{VIEW}, v_p, -\rangle$ message directly from c_p, but from a process $p_1 \in \overline{v_p}$. Since the $\langle \text{VIEW}, v_p, -\rangle$ message has been m-received after the m-sending of the PROPOSE message for the agreement on v_q. By monotony of $estimate$ process, p_1 must belong to $\overline{v_q}$. Thus, p_1 has participated in the construction of both v_p and v_q by m-sending two PROPOSE messages to c_p and c_q in this order. And, it has m-received a VIEW message containing v_p after the m-sending of the PROPOSE message for the agreement on v_q. By iterating the reasoning, we obtain an unbound chain $p_1, \ldots, p_n, \ldots$ of processes belonging to $\overline{v_p}$. This is a contradiction. The cardinality of $\overline{v_p}$ is finite and all these processes are different (each process m-sends the same VIEW message at most once). Now, suppose p has m-sent the PROPOSE message to c_q before the PROPOSE message to c_p. By monotony of $estimate$, q must have m-sent the PROPOSE messages to c_p and c_q in the same order. But, this leads to a contradiction as in the previous case and concludes the proof of the theorem. □

The next property to prove is View Completeness. As the reader can note, its proof is simpler than the previous one since if a process p m-suspects another process q, p will permanently remove q from its view estimate and no process can force p to insert q again.

Theorem D.2 (View Completeness). *The PGMS algorithm satisfies Property GM2.*

Proof. Let Θ be a set of processes such that there is a time after which all processes in Θ are unreachable from $\Pi - \Theta$. Let p be a correct process in Θ and let q be a process in $\Pi - \Theta$. We must prove that there is a time after which the current view of p never contains q. Suppose p installs a bound number of views. The last view installed by p cannot contain q, otherwise p will enter agreement phase again and, by Corollary D.1, it will install a new view after the last. Thus, suppose p installs an unbound sequence of views. By Property A.1.1, there is a time t_1 after which p permanently m-suspects q. This implies that, after t_1, q is permanently excluded from $reachable_p$. This implies that there is a time $t_2 \geq t_1$ after which all messages $\langle \text{PROPOSE}, s_p\rangle$ m-sent by p during a view are such that $q \notin s_p.estimate$. Thus, all views installed by p after t_2 cannot contain q. □

Now, we must prove that our algorithm satisfies View Order and View Integrity.

Theorem D.3 (View Order). *The PGMS algorithm satisfies Property GM4.*

Proof. Let v, w be two views. We must prove that if $v \prec^* w$, then $w \not\prec^* v$. By contradiction, suppose there exist two chains of views, $v \equiv v_1 \prec_{p_1} v_2 \prec_{p_2} \ldots \prec_{p_{n-1}} v_n \equiv w$ and $w \equiv v_{n+1} \prec_{p_{n+1}} v_{n+2} \prec_{p_{n+2}} \ldots \prec_{p_{m-1}} v_m \equiv v$. Let t_i denote the time at which the coordinator c_i generated the VIEW message containing v_i and let t'_i be the time at which p_i installed v_i. Obviously, $t_i < t'_i$. When p_i m-receives the VIEW message that contains v_{i+1}, it verifies that the agreement on v_{i+1} has been reached with information m-sent by p_i in v_i. Therefore, $t'_i < t_{i+1}$. Thus, $t_i < t_{i+1}$. By transitivity, we can state that $t_1 < t_n$ and that $t_n < t_m = t_1$, a contradiction that concludes the proof. □

Theorem D.4 (View Integrity). *The PGMS algorithm satisfies Property GM5.*

Proof. Let p be a process. We must prove that each view installed by p contains p itself. The first view installed by p is equal to $\{p\}$. Before installing any other view v, p must m-receive a message $\langle \text{VIEW}, w, C\rangle$. By Property A.1.1, p belongs to the destination set of the VIEW message. By construction, this set coincides with $C[p].estimate$. Since v contains all processes in $C[p].estimate$ that have m-received the VIEW message containing the predecessor of v at p, then p belongs to $\overline{v}$□.

Finally, we must prove that our algorithm satisfies the View Coherency property. First of all, we prove that a process will eventually enter agreement phase if a process in its current view v never installs v. Then, we use this lemma to show that our algorithm satisfies the three parts of View Coherency.

Lemma D.5. *If a correct process p installs a view v, then, for every process $q \in \overline{v}$, either 1) q also install v or 2) p will eventually enter agreement phase during v.*

Proof. By contradiction, suppose q never installs v and p never enters agreement phase during v. Thus, p never m-suspects q after the installation of v. By Property A.1.1, q is correct. By Property A.1.5, q will eventually m-receive the VIEW message forwarded by p before installing v. Since q does not install v, there are two possibilities: q discards the VIEW message containing v either because it has installed a new view or because p does not belong to $estimate_q$. In both cases, q must have excluded p from $estimate_q$ after m-sending the PROPOSE message with which the agreement on v was reached. After the exclusion, q has installed a view w not containing p. By Property A.1.7, there is a time after which q stops m-suspecting p. Then, q will eventually enter agreement phase and m-send a SYNCHRONIZE message to p. By Property A.1.5, p will eventually m-receive the message and enter agreement phase during v, a contradiction that concludes the proof. □

Theorem D.5 (View Coherency). *The PGMS algorithm satisfies Property GM3.*

Proof.

1. Let v be a view installed by a correct process p and let q be a process in $\overline{v}$ that never installs v. The proof follows from Lemma D.5 and Corollary D.1.
2. Let v be a view installed by two processes p and q, and suppose p changes view after having installed v. If q is correct, we must prove that q will eventually install an immediate successor to v. By Corollary D.1, it is sufficient to prove that p will eventually enter the agreement phase during v. Suppose q stops m-suspecting p after having installed v (otherwise the proof is trivial). By Property A.1.1, p is correct. By Property A.1.7, there is a time t_1 after which p stops m-suspecting q. There are two possibilities:
   - If p installs a bound number of views, let w be the last view installed by p. By hypothesis, $v \prec^* w$. Process q must belong to $\overline{w}$, since otherwise p would enter agreement phase again and install a new view. By Lemma D.5 and Theorem D.3, q will install w after v (by hypothesis p does not install other views after w) and the lemma is proven.
   - If p installs an unbound number of views, q will m-receive an unbound number of SYNCHRONIZE messages m-sent by p after t_1. This implies that q enters the agreement phase during v.
3. Let p be a process that installs a view v as well as its immediate successor w, both containing q. We must prove that p installs w only after q has installed v. Note that $\overline{w}$ is obtained from the last VIEW message by excluding all processes in $\overline{v}$ whose last view is different from v. Since q belongs to both $\overline{v}$ and $\overline{w}$, the last view of q must be equal to v. □

REFERENCES

[1] Y. Amir et al. "The Totem Single-Ring Ordering and Membership Protocol," *ACM Trans. Computer Systems,* vol. 13, no. 4, pp. 311-342, Nov. 1995.

[2] Y. Amir and J. Stanton, "The Spread Wide-Aread Group Communication System," technical report, Center of Networking and Distributed Systems, Johns Hopkins Univ., Baltimore, Md, Apr. 1998.

[3] E. Anceaume et al. "On the Formal Specification of Group Membership Services," Technical Report TR95-1534, Computer Science Dept., Cornell Univ., Ithaca, N.Y., Aug. 1995.

[4] T. Anker et al. "Scalable Group Membership Services for Novel Applications," *Proc. DIMACS Workshop Networks in Distributed Computing,* pp. 23-42, 1998.

[5] Ö. Babaoglu et al. "RELACS: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems," *Proc. Hawaii Int'l Conf. System Sciences,* pp. 612-621, Jan. 1995.

[6] Ö. Babaoglu et al. "System Support for Partition-Aware Network Applications," *Proc. Int'l Conf. on Distributed Computing Systems,* pp. 184-191, Amsterdam, May 1998.

[7] K. Birman, "The Process Group Approach to Reliable Distributed Computing," *Comm. ACM,* vol. 36, no. 12, pp. 36-53, Dec. 1993.

[8] K. Birman and R. van Renesse, *Reliable Distributed Computing with the ISIS Toolkit.* Los Alamitos, Calif.: IEEE CS Press, 1994.

[9] T. Chandra et al. "On the Impossibility of Group Membership," *Proc. ACM Symp. Principles of Distributed Computing,* pp. 322-330, May 1996.

[10] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM* vo. 43 no.1, pp. 225-267 Mar. 1996

[11] D. Dolev et al. "Failure Detectors in Omission Failure Environments," *Proc. ACM Symp. Principles of Distributed Computing,* Aug. 1997.

[12] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communication," *Comm. ACM,* vol. 39, no. 4, Apr. 1996.

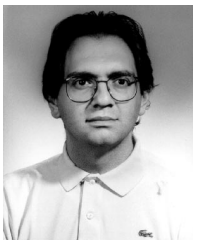
[13] D. Dolev, D. Malki, and R. Strong, "An Asynchronous Membership Protocol that Tolerates Partitions," Technical Report CS94-6, Inst. of Computer Science, Hebrew Univ. of Jerusalem, Mar. 1994.

[14] D. Dolev, D. Malki, and R. Strong, "A Framework for Partitionable Membership Service," Technical Report CS95-4, Inst. of Computer Science, Hebrew Univ. of Jerusalem, 1995.

[15] D. Dolev, D. Malki, and R. Strong, "A Framework for Partitionable Membership Service," *Proc. ACM Symp. Principles of Distributed Computing,* May 1996.

[16] P.E. Ezhilchelvan, R.A. Macêdo, and S.K. Shrivastava, "Newtop: A Fault-Tolerant Group Communication Protocol," *Proc. Int'l Conf. Distributed Computing Systems,* June 1995.
[17] A. Fekete, N. Lynch, and A. Shvartsman, "Specifying and Using a Partitionable Group Communication Service," *Proc. ACM Symp. Principles of Distributed Computing,* Aug. 1997.
[18] M.J. Fischer, N.A. Lynch, and M.S. Patterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374-382, Apr. 1985.
[19] R. Friedman and R. van Renesse, "Strong and Weak Virtual Synchrony in Horus," Technical Report TR95-1537, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Mar. 1995.
[20] M. Hayden, "The Ensenble System," doctoral dissertation, Computer Science Dept., Cornell Univ., Ithaca, N.Y., Jan. 1998.
[21] F. Kaashoek and A. Tanenbaum, "Group Communication in the Amoeba Distributed Operating System," *Proc. IEEE Symp. Reliable Distributed Systems,* pp. 222-230, May 1991.
[22] R. Khazan, A. Fekete, and N. Lynch, "Multicast Group Communication as a Base for a Load-Balancing Replicated Data Service," *Proc. Int'l Symp. Distributed Computing,* Sept. 1998.
[23] C. Malloth, "Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large-Scale Networks," doctoral dissertation, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 1996.
[24] C. Malloth and A. Schiper, "View Synchronous Communication in Large Scale Networks," *Proc. Open Workshop ESPRIT Project Broadcast,* July 1995.
[25] A. Montresor, "A Dependable Registry Service for the Jgroup Distributed Object Model," *Proc. European Reasearch Seminar Advances in Distributed Systems (ERSADS '99),* Apr. 1999.
[26] A. Montresor, "The Jgroup Reliable Distributed Object Model," *Proc. IFIP Int'l Working Conf. Distributed Applications and Systems,* June 1999.
[27] L. Moser et al. "Totem: A Fault-Tolerant Group Communication System," *Comm. ACM,* vol. 39, no. 4, Apr. 1996.
[28] L.E. Moser et al. "Extended Virtual Synchrony," *Proc. Int'l Conf. Distributed Computing Systems,* June 1994.
[29] G. Neiger, "A New Look at Membership Services," *Proc. ACM Symp. Principles of Distributed Computing,* May 1996.
[30] R. De Prisco et al. "A Dynamic View-Oriented Group Communication Service," *Proc. ACM Symp. Principles of Distributed Computing,* June 1998.
[31] A. Ricciardi and K. Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments," *Proc. ACM Symp. Principles of Distributed Computing,* pp. 341-352, Aug. 1991.
[32] A. Schiperand and A. Ricciardi, "Virtually-Synchronous Communication Based on a Weak Failure Suspector," *Proc. Int'l Symp. Fault-Tolerant Computing,* pp. 534-543, June 1993.
[33] R. van Renesse et al. "The Horus System," *Reliable Distributed Computing with the Isis Toolkit,* pp. 133-147, Los Alamitos, Calif.: IEEE CS Press, 1993.
[34] R. van Renesse, K.P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Comm. ACM,* vol. 39, no. 4, pp. 76-83, Apr. 1996.

Özalp Babaoglu received a PhD degree from the University of California at Berkeley in 1981 where he was one of the principal designers of BSD Unix. He is a professor of computer science at the University of Bologna, Italy. Before moving to Bologna in 1988, Dr. Babaoglu was an associate professor in the Department of Computer Science at Cornell University. He is active in several European research projects exploring issues related to fault tolerance and scale in distributed systems. He serves on the editorial boards for for *ACM Transactions on Computer Systems* and *ACM Springer-Verlag Distributed Computing*.



Renzo Davoli (M-91) received his degree in mathematics from the University of Bologna (Italy) in 1986. In 1991, he joined the Department of Mathematics of the same university as a research associate. He has been a member of the Computer Science Department since its founding in 1995, where he is currently an associate professor of computer science. His research interests include large-scale distributed systems, real-time systems, nomadic computing, wireless systems, and neural networks. Dr. Davoli is a member of the IEEE Computer Society, ACM, and AICA.

Alberto Montresor received the MS degree in computer science in 1995 and the PhD degree in computer science in 2000, both from the University of Bologna. He currently holds a postdoctoral position at the University of Bologna. His research interests include distributed computing, fault tolerance, and distributed object frameworks.