Process Membership in Asynchronous Environments

Aleta M. Ricciardi*
Kenneth P. Birman*

TR 93-1328
(replaces TR 91-1188)
February 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Authors supported by DARPA/NASA Ames grant NAG 2-593 and by grants from IBM and Siemens Corporation.

Process Membership in Asynchronous Environments

Aleta M. Ricciardi,
Kenneth P. Birman*
Cornell University
Department of Computer Science
Ithaca, NY 14853-7501 USA
aleta@cs.cornell.edu, ken@cs.cornell.edu
FAX 607-255-4428

February 9, 1993

Abstract

The development of reliable distributed software is simplified by the ability to assume a fail-stop failure model. We discuss the emulation of such a model in an asynchronous distributed environment. The solution we propose, called Strong-GMP, can be supported through a highly efficient protocol, and has been implemented as part of a distributed systems software project at Cornell University. Here, we focus on the precise definition of the problem, the protocol, correctness proofs, and an analysis of costs.

Keywords Asynchronous computation; Fault detection; Process membership; Fault tolerance; Process group.

*Authors supported by DARPA/NASA Ames Grant NAG 2-593, and by grants from IBM and Siemens Corporation.

1 Introduction

The development of distributed software is greatly simplified in environments where process and communication failures are benign. For this reason, it is common for distributed systems to be developed under the assumption that the communication network does not partition and that processes are fail-stop [19, 20] – that they fail only by halting, and that these failures are detected accurately.

Unfortunately, real distributed environments are not entirely benign in these respects. On the one hand, the assumption that programs fail by halting can be satisfied to a good approximation by careful development methodology and testing. Similarly, most communication failures, such as message loss, corruption, out-of-order delivery, and replay, can be detected and corrected at low cost, again with high probability. However, this is not the case for failure detection and network partitions. Communication partitions are unavoidable in networks, and when they occur, may mimic process failures.

It is well known that the consensus problem cannot be solved in asynchronous systems subject to process failures [10], and this is often taken to mean that software for realistic environments must live with some risk of inconsistent failure detection. A related result exists for the database commit problem in the presense of partition failures [21]. A consequence is that a great deal of the 'fault-tolerant' distributed software used in contemporary networks is at risk of some form of inconsistent or incorrect behavior when an action is based on the apparent detection of a process failure.

That such inconsistencies are not very noticeable testifies to the ingenuity of systems developers in building systems for which inconsistency is not a fatal condition, but also to the extremely limited use of genuinely distributed programs in modern networks. Most distributed software is based on one-time interactions between a client program and a server; it is very uncommon to see distributed systems in which any form of continually evolving distributed state is shared among multiple processes. In client-server systems, it is uncommon that the detailed behavior of different programs would be compared; hence, inconsistencies in how programs report and react to failures might not affect the 'distributed' computation, much less be noticed by a casual observer.

Unfortunately, the need to develop fault-tolerant distributed software with non-trivial distributed state in modern computer networks is seen more and more often in modern computer applications. One of us (Birman), through work with a distributed programming environment called Isis [6, 5], has gained experience with a wide range of complex distributed applications in settings such as telecommunications, factory automation, finance, scientific computing and the military. In these domains one finds problems that are inherently distributed and require fault-tolerance, and also in which complex distributed state is needed to operate the desired system correctly. For example, a telecommunications system must react to failures of switching nodes in a consistent manner; inability to do this can cause the system to deny service. A brokerage system may need to provide trading advice, based on changing market conditions, to multiple traders. If two analytic servers are started because some parts of the system incorrectly sense a primary server as having failed, different traders may be given differing, inconsistent advice. In settings such as these, inconsistent behavior can have significant implications and cannot be tolerated.

Similarly, modern distributed operating systems exhibit features that require accurate failure detection. For example, the Mach operating system [15] identifies communication endpoints using an abstraction called the *communication port.* Each port has a single "receive right", bound to one process. Rights to send data to a port can be passed among processes, and are carefully tracked by Mach. Mach guarantees that communication to a port will be reliable: if a successful outcome is reported to the sender, the message will not be lost unless the destination fails, and a failure is reported only if the destination is faulty. Additionally, Mach notifies the holder of a receive-right when all holders of send rights have deleted them (or failed), and notifies the holder of a send right if the corresponding receiver fails.

Mach is widely cited for its simple and powerful communications model, and has emerged as an industry standard. However, it is easy to see that this model cannot be implemented in a way that is both safe and live: the only "safe" way to detect a failure is to wait for the faulty process to restart, and this can introduce unbounded delays! At the time of this writing, Mach waits for failed nodes to restart before reporting failures, hence even a single failure could prevent the system from making progress.

Our work proposes an approach which, although subject to limitations stemming from the impossibility results cited above, is nonetheless extremely powerful. The basic idea is to substitute a *logical* notion of system membership for the *physical* notion of "operational" or "failed". In our scheme, application programs define operational processes to be those listed by the membership service and failed processes to be those not listed as members of the system. To the extent that the membership service is able to report consistent information to processes using it, those processes can then implement consistent, fault-tolerant distributed algorithms.

Our membership service assumes a low-level mechanism that monitors the status of processes. The membership service excludes any process from the system that this mechanism *suspects* of having failed. If the removed process has not crashed (i.e. the suspicion was incorrect or due to a transient condition that corrected itself), subsequent communication from it to the remainder of the system is inhibited. In this way we prevent a "zombie" process from contradicting the abstraction presented to the remaining system processes. Lastly, a faulty process that recovers will be notified that it has been dropped from the system.

When physical partitions occur, our membership service prevents the system from logically partitioning. More precisely, our scheme distinguishes the *majority* partition from *minority partitions.* By defining the state of the majority partition to be the true system state and limiting the actions permitted in a minority partition, logically consistent behavior can be guaranteed even when a partition occurs. During periods when a majority partition cannot be constituted, our scheme might treat all partitions as minority ones, effectively halting the system. The approach is thus one that provides rigorously consistent behavior at all times, although it may not permit progress in infrequent situations caused by severe network partitions.[1]

As an example, our membership service could be used to overcome the failure detection problems currently encountered in Mach. Mach could be made both safe and live if it were modified to 1) report 'apparent' failures to our service (thereby making Mach one of our suspector mechanisms), 2) treat machines and processes as faulty only when our service reported them as such, and 3) restrict communication to members of the system (as defined by our membership service). The Mach communication guarantees would then be satisfied even in networks where transient disruptions would sometimes cause Mach to suspect failures incorrectly. We believe that most application developers would prefer the environment our service provides to one that could incur indefinite delays.

Agreement on the membership of a group of processes in a distributed system is a classic problem, and has been treated elsewhere. Relevant prior research includes solutions for database contexts [4], real-time settings [8], and distributed control applications [12, 5]. Cristian [9], specified and solved a problem similar to the one we consider here, but in contrast to our work, he considered a synchronous setting. Our approach and solution focus on the asynchronous case, but differ from previous work on group membership for asynchronous systems. The membership semantics provided by Virtual Partitions [1] are weaker, allowing multiple membership views to exists simultaneously, and requiring neither atomicity nor uniformity in committing new views. These semantics however reflect a desire to maintain replicated data availability; our goal is to provide a consistent, unique source of system-wide membership information. In contrast to [13, 2], which also permit multiple membership

[1] Through experience with several hundred Isis applications, we have observed that the most common partition case involves a single processor disconnected from its LAN. Network "bridge" failures are uncommon in LAN settings, and it makes sense to treat WAN systems differently from LAN's because a WAN has different performance characteristics. Other systems have adopted different approaches to this issue, however, such as in Dolev's Transis project [3].

views, we do not assume the existence of an underlying fault-tolerant atomic, ordered multicast. The protocol of Mishra, et.al. [14] also relies on an ordered multicast. In these cases, the potential membership must be a static set of processes so that the multicast ordering properties can be maintained. This makes handling process recoveries more straightforward, but still requires additional mechanism to join newly-created processes. We consider only point-to-point communication and an arbitrary, unknown set of system processes. We handle joins arising from both process recovery and process creation with the same mechanism. The protocol in Birman and Joseph [5] blocks during periods when failures and recoveries occur continuously. Our solution is fully 'online': we can process a constant flow of requests to both remove and add processes, which is exactly what occurs in actual systems.

In Section 2 we describe our system model and the formal language we will use to specify the Strong Group Membership Problem (Strong GMP). In Section 3 we specify Strong GMP, and in Section 4 we present our solution, the S-GMP algorithm. Section 5 gives the main part of the inductive correctness proof and discusses the protocol's message complexity and minimality. We conclude by discussing the implications of our particular specification, and directions for future work.

2 The System Model and Formal Logic

We consider only asynchronous distributed systems in which processes fail by crashing. *Distributed* means that the processors are physically separated and that processes executing in the system communicate only by passing messages along a fixed set of channels. *Asynchronous* means that the system has no global clock, and that there are no bounds on relative local clock speeds, execution speeds, or message transmission delays. The asynchrony assumption is realistic: system load, network traffic, and any other dynamic components of the system that affect performance all conspire to violate synchronization assumptions.

Before defining the abstract computational model, we discuss the goals and effect of our membership service for processes in asynchronous systems.

2.1 Membership Service Goals

This paper focuses on the events that occur at processes after the membership service is already established (Section 4.2.4 discusses cold-starting the service). Our goal in building this membership service is to provide processes in an asynchronous system subject to halting failures, with an execution environment indistinguishable from a synchronous, halting-failure system. Here, the term "indistinguishable" refers to the sequence of events observed by a process *while it is a member of the system.* The situation for a process excluded from the system is discussed below.

Our solution has the property that when a process, p, learns from the membership service that another process, q, is no longer a member of the system, p can identify an event in its execution after which it will never receive another message from q. For p, this is indistinguishable from q crashing and the membership service detecting it accurately. Moreover, our solution constructs a *consistent cut* [7] along which every other functioning member of the system will also learn that q is excluded. Consequently, p can take actions that depend both on q having crashed and on all other processes learning this concurrently (just as it could in a synchronous environment). In a normal asynchronous system, p would have neither guarantee.

In our model, a new process, p, must *join* the system via the membership service before it can interact with other processes. The service responds with the current *system membership list*, and thereafter keeps p informed of each change to the list. For as long as p remains on the list, it can send messages to all other listed processes, and communication appears to be reliable and FIFO.[2] Finally, our work has the property that all members of the system observe exactly the same sequence of membership changes (join and leave events), even when members of the membership service itself fail or join. Elsewhere [18] we show how this strong, same-sequence property both simplifies distributed algorithms that take actions based upon membership changes, and, somewhat paradoxically, actually helps in reducing the cost of the membership service protocol itself.

Processes that genuinely *fail* do so by halting. We require that such a process is eventually suspected of having failed, and then removed from the system list.[3] Of course, no live failure detection protocol for asynchronous systems can avoid mistakenly suspecting an operational process and then removing it from the membership list [10]. Because exclusion from the membership list will be equated with failure, such exclusions must result in executions that are consistent with those in which the excluded process had, in fact, failed. Specifically, we must suppress communication from a process that has been erroneously excluded. To this end, in addition to FIFO and channel reliability assumptions, we assume processes sever

[2]It is well know that an underlying message transport system that uses sequence numbers, acknowledgements, and retransmission can overcome message loss, duplication, and out-of-order arrival.

[3]While we are not concerned with the implementation of the failure suspicion module, this can be quite inexpensive. In particular, because our membership service places a uniformly observed ranking on system members it is not necessary that every process monitor every other process. For example, the scheme used in the Isis system requires each process to monitor only the next-highest ranked process. This seems to imply a linear cost, but because network speeds are very high the dominant cost turns out to be the overhead imposed on processes, which is constant and unrelated to system size in this case.

communication paths with all others they believe faulty.[4]

From the excluded process's, say q's, point of view, it can no longer communicate with other processes, but it can continue local computations. To illustrate the issues suppose q had been the token-holder in a protocol that orders multicasts among a subset, S, of processes. Upon learning of q's failure, the remaining processes in S determine a new token holder, say q', although q will continue believing it is the token holder. Since q can no longer make its message ordering known to S , the fact that q's and q''s orderings may differ does not violate the (observable) correctness of the message-ordering protocol. That q will 'observe' a different ordering than the rest of S is irrelevant.

2.2 System Requirements and Model Assumptions

To implement the FIFO and channel reliability properties we require two things of the physical system. First, each message sent along a channel must have a non-zero probability of reaching its destination intact, and second, each process must have a local, monotonically increasing clock (i.e. counter). These two requirements suffice to implement live failure suspectors, and a completely-connected network of reliable, FIFO channels. Our protocols will assume this complete package of communication guarantees, but we are not concerned with how they are implemented.

As soon as one processes, p, suspects another of having failed, it *Disconnects* all its communication channels with the suspected process. Moreover, to hide as quickly as possible an erroneous suspicion, p *Gossips* (for example, with piggy-backs) its suspicion to all other processes in further communication, whereupon recipients adopt p's belief and also disconnect themselves from the suspected process. The Gossip and Disconnect actions combine to *isolate* suspected-faulty processes among processes not believing each other faulty.

The [10] impossibility result can be interpreted as forcing applications in asynchronous systems to either make accurate failure detections or be live. By choosing liveness, we admit the possibility of erroneous failure detections, but by isolating mistakenly suspected processes, we prevent them from further affecting the global system. As a result, q halting and q mistakenly suspected to have halted are indistinguishable.

[4]Because the communication layer is asynchronous, messages from an excluded process may continue to arrive, and be rejected, for an unbounded period of time. The communication layer would also inform an excluded process that it has been excluded, causing it to rejoin the system under a new process identifier. The protocols needed to implement such a transport layer are evident and will not be presented here.

2.3 The System Model

Denote by Proc a countable set of process identifiers, $\{p_1, p_2, \ldots\}$. The process name space is infinite so that we can model infinite executions in which new processes continually arise. However because there can be only finitely-many processors, and because process births require non-zero time, the number of processes extant at any real time in an execution will always be finite.

Processes may send and receive messages, and do internal computation. The event $send_p(q, m)$ denotes p sending message m to q, and $recv_q(p, m)$ denotes q's receipt of m from p. The distinct internal event $crash_p$ models the crash failure of process p, after which only other $crash_p$ events are permitted. A *history* for process p, denoted h_p, is a sequence of events executed by p, and must begin with the distinct, internal event $start_p$:

$$h_p \stackrel{\text{def}}{=} \left\langle start_p \cdot e_p^1 \cdot e_p^2 \cdots e_p^k \right\rangle \qquad k > 0.$$

We write $e \in h_p$ when e is an event of h_p. A *cut* is an n-tuple of process histories $c = (h_{p_1}, h_{p_2}, \ldots, h_{p_n})$, where $p_i \in$ Proc. We restrict our attention to cuts determined by finite subsets of Proc since these represent the system's global system state at some real time in its execution. Each execution begins with the distinct cut, $c_0 = < start_{p_1}, start_{p_2}, \ldots, start_{p_n} >$. We also write $e \in c$ to abbreviate "$e \in h_p$ for some p mentioned in c", and elaborate when the context does not clearly distinguish the intention.

We assume familiarity with the *happens before* relation [11] between events (written $e \rightarrow e'$), and also with *consistent cuts* [7]. Henceforth we restrict the discussion to consistent cuts, as they are the ones that are physically realizable. Consistent cuts are the possible global states of an execution; while a given consistent cut may never have existed at any point in real time, it is impossible for a cut that is not causally consistent to ever exist at any point.

A characterization of global causality should incorporate the notion of progress between global states. Specifically, we desire that every process either makes local progress or remains stationary, none should regress. A process makes local progress between the cumulative states represented by h_p and h'_p exactly when h_p is a prefix of h'_p.

Definition Given $c = (h_1, \ldots, h_p, \ldots, h_n)$ and $c' = (h'_1, \ldots, h'_p, \ldots, h'_n)$, *$c$ causally precedes c'* (written $c \leq c'$) if and only if for each process, p either **1)** $h_p = h'_p$; or **2)** h_p is a strict prefix of h'_p.

Observe that there are (infinitely) many completions for any given cut. In this sense, the future of any cut is uncertain; it may branch out in many directions. On the other hand, $c \leq c'$ implies that any execution in which c' is a prefix must also contain c as a prefix.

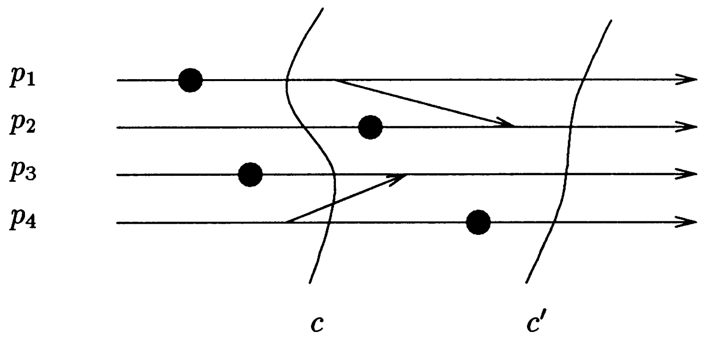


Figure 1: h_{p_i} is a strict prefix of h'_{p_i} for each p_i so $c \ll c'$.

Definition Let $c = (h_1, \ldots, h_p, \ldots, h_n)$ and $c' = (h'_1, \ldots, h'_p, \ldots, h'_n)$ be consistent cuts. Then c *strictly precedes* c' (written $c < c'$) if and if $c \leq c'$ and $c \neq c'$; the cut c *very strictly precedes* c' (written $c \ll c'$) if and only if h_p is a strict prefix of h'_p for each p mentioned (Figure 2.3). ∎

2.4 The Modal Logic

So far, our description of Strong GMP refers to *when* core members agree on the group view, as well as the degree of *simultaneity* with which they do so. A *temporal modal logic* allows us to express these notions. Unique to our logic is its attention to asynchrony - the basic semantic entities of the logic are consistent cuts. We briefly describe the temporal modalities we use to specify Strong GMP.

Given a propositional formula, ϕ, and the $\leq$ relation between cuts, the formula $\Box\phi$ holds along cut c precisely when ϕ holds along *all* future cuts in all runs containing c (i.e. every c' such that $c \leq c'$). $\Diamond\phi$ holds along c when ϕ will hold along *some* future cut in every run containing c. We interpret $\Diamond$ as "inevitability". $\ominus\phi$ holds along c if ϕ held at some $c' \leq c$, and $\boxminus\phi$ if ϕ held along all $c' \leq c$.

3 Strong Group Membership

We now formally define the Strong Group Membership Problem for asynchronous systems. Our definition specifies how to coordinate local events among a group of processes so that the group's externally observed behavior is indistinguishable from that of a single, fault-tolerant process. Thus, any solution to Strong GMP can be used to build a system membership service (which we call a Membership Resource Manager, or MRM). In this section, and in the rest of the paper, we restrict our focus to the *core* processes implementing the MRM – the formal problem describing their actions, and the algorithm solving this problem. Thus, we describe a *hierarchical* approach to building a Strong GMP membership service, in that our protocol is run by a small core set of processes, which use a cheap replication scheme (e.g. the Isis replication tools) to maintain a fault-tolerant member list for the overall system.

Creating the illusion of a single fault-tolerant process means that core members must agree, not only on the entire system membership, but also on the composition of the MRM core. A core member that fails or is otherwise removed must be consistent with the rest of the core while it is a member. More important, a core member that is removed from the core but has not halted must not be able to misrepresent the system state; our specification must preclude such a process from changing its local view of the system's membership or the core's membership independently.

3.1 Formal Specification

The formula $\textsc{up}_p$ holds along a cut if and only if p has not executed $crash_p$ in its local history component of that cut. Conversely, $\textsc{down}_p$ holds along c exactly when p has crashed in c. The indexical set $\mathsf{Up}(c)$ in an asynchronous run A is the set of all processes that have not crashed along c : $\mathsf{Up}(c) = \{p \mid \textsc{up}_p \text{ holds along } c\}$.

Process p executes the event $faulty_p(q)$ as soon as it suspects q faulty; whether p comes to suspect q through some local observation or through our Gossip assumption (Section 2.2) is immaterial. Some time after recording $faulty_p(q)$, p will execute the event $remove_p(q)$. The distinction between these events is significant: $faulty_p(q)$ represents p's belief in q's faultiness, which may be incorrect, while $remove_p(q)$ is actual removal of q from the set of core members p believes operational. The formula $\textsc{faulty}_p(q)$ holds along all cuts that include $faulty_p(q)$, and $\textsc{remove}_p(q)$ along all cuts that include $remove_p(q)$. Analogous statements hold for events $operating_p(q)$ (p believes q is functional) and $add_p(q)$ (p adds q to the set of core members), and formulas $\textsc{operating}_p(q)$ and $\textsc{add}_p(q)$. In contrast to $\textsc{faulty}_p(q)$, $\textsc{operating}_p(q)$ is not stable.

The *local membership view* for process p along cut $c = (h_1, \ldots, h_p, \ldots, h_n)$, (denoted $\mathsf{LocalView}_p(c)$), is the set of processes p obtains by sequentially modifying its initial membership list according to the $remove_p()$ and $add_p()$ events in h_p. We use $\mathsf{LocalView}_p$ when the cut is clear from context. Trivially, we require $p \in \mathsf{LocalView}_p(c)$. The formula $\textsc{in-local}_p(q)$ holds along all cuts, c, such that $q \in \mathsf{LocalView}_p(c)$. Because h_p is linear, it makes sense to talk about the x^{th} version of p's local view, which we denote $\mathsf{LocalView}_p^x$. Finally $\textsc{in-local}_p^x(q)$ holds when $q \in \mathsf{LocalView}_p^x$. The formula $\textsc{NotDef'd}(\mathsf{LocalView}_p^x)$ holds along c if p has not (yet) defined it's x^{th} local view.

We extend local views to *group views* as follows. Given $\mathsf{S} \subseteq \mathsf{Proc}$, and a consistent cut c, if the local views of all the functional processes in S are identical, the group view is the agreed-upon local view; if S has no functioning members or if the functioning members of S have different local views, the group view is undefined. We say that S *determines* a group view. Formally :

Definition Given a consistent cut c and a set of processes, $\mathsf{S} \subseteq \mathsf{Proc}$, the *group view determined by S along c* is :

$$\mathsf{GpView}_{\mathsf{S}}(c) = \begin{cases} \mathsf{LocalView}_p(c) & \bigwedge\Big(p, q \in \big(\mathsf{S} \cap \mathsf{Up}(c)\big) \neq \emptyset\Big) : \\ & \Big(\mathsf{LocalView}_p(c) = \mathsf{LocalView}_q(c)\Big) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

∎

The formula UNDEF'D($\mathsf{GpView}_{\mathsf{S}}(c)$) holds along c if the local views of any functional members of S disagree or if $\mathsf{S} \cap \mathsf{Up}(c) = \emptyset$.

Constraining Membership in $\mathsf{GpView}_{\mathsf{S}}(c)$

The definition of $\mathsf{GpView}_{\mathsf{S}}(c)$ is crucial to the class of Group Membership Problems so it is worthwhile discussing how the sets S and $\mathsf{GpView}_{\mathsf{S}}(c)$ relate. Recall that $\mathsf{GpView}_{\mathsf{S}}(c)$ is the abstraction we are using to define the single, fault-tolerant process illusion that will be used to build an MRM.[5] In this light, MRM core members are precisely the members of $\mathsf{GpView}_{\mathsf{S}}(c)$.

If $q \in \big(\mathsf{GpView}_{\mathsf{S}}(c) \cap \overline{\mathsf{S}}\big)$ then q is a core MRM member whose local view is not used in determining either the MRM composition or the total system membership. Specifically, q's local view is not constrained by the definition of $\mathsf{GpView}_{\mathsf{S}}(c)$, so $\mathsf{LocalView}_q(c)$ need not be identical to $\mathsf{GpView}_{\mathsf{S}}(c)$. Because q replies to MRM client requests based on its local view, its replies will contradict other core members' replies when $\mathsf{LocalView}_q(c) \neq \mathsf{GpView}_{\mathsf{S}}(c)$; the single-process illusion falls apart. Consequently, unless every core member's local view is used to determine the MRM group view, the MRM cannot guarantee global consistency.

To avoid this, our specification forces q to be in S whenever it is in $\mathsf{GpView}_{\mathsf{S}}(c)$:

$$\Big(\mathsf{GpView}_{\mathsf{S}}(c) \cap \mathsf{Up}(c)\Big) \subseteq \Big(\mathsf{S} \cap \mathsf{Up}(c)\Big).$$

[5]In practice, the group view, and therefore each core member's local view, includes the entire system membership in addition to the MRM composition; here we are only concerned with the MRM composition.

The reverse inclusion follows trivially since $p \in \mathsf{LocalView}_p(c)$. Consequently, our specification requires $\mathsf{S} = \mathsf{GpView}_{\mathsf{S}}(c)$.

To finish the single-process illusion, the MRM must be unique. We will therefore also require that there be at most one set satisfying this equality along any consistent cut. Since there can only be one MRM, some form of quorum consensus is needed to change the global system membership. If a quorum cannot be attained (for example during certain partitions), no solution to Strong GMP can make progress.

Finally, $\mathsf{GpView}_{\mathsf{S}}(c)$ is defined if and only if the local views of all its functioning members agree. Processes that are eventually removed from the core are not excused from having consistent views while they are members. Moreover, a core member that is removed but has not crashed cannot be allowed to change its local view. Our specification captures these safety issues in two clauses: GMP-2 formalizes the *Uniqueness* requirement for group views, and GMP-3, by requiring every local view to exist as a group view, prevents excluded processes from taking actions unilaterally.

3.2 Strong GMP Specification

We now have the language necessary to formalize Strong GMP. Since formulas are evaluated along cuts we drop references to cuts in indexical sets.

GMP-0 (Base Case) An initial group view eventually exists:

$$\Diamond \bigvee_{\mathsf{S}_0 \subseteq \mathsf{Proc}} \left(\mathsf{S}_0 = \mathsf{GpView}_{\mathsf{S}_0}()\right).$$

GMP-1 (Validity) Processes do not make changes to their local views capriciously. For example, if q were once, but is not currently, in $\mathsf{LocalView}_p$, then p should believe q faulty.

a. $\left(\diamondminus \text{IN-LOCAL}_p(q) \wedge \neg\text{IN-LOCAL}_p(q)\right) \Rightarrow \text{FAULTY}_p(q)$

b. $\left(\diamondminus \neg\text{IN-LOCAL}_p(q) \wedge \text{IN-LOCAL}_p(q)\right) \Rightarrow \diamondminus\ \text{OPERATING}_p(q).$

In contrast to $\text{FAULTY}_p(q)$, $\text{OPERATING}_p(q)$ is not stable.

GMP-2 (Uniqueness) Non-null group views are unique along all consistent cuts.

$$\bigvee_{\mathsf{S} \subseteq \mathsf{Proc}} \left(\mathsf{GpView}_{\mathsf{S}}() = \mathsf{S}\right) \Rightarrow \bigwedge_{\emptyset \neq \mathsf{S}' \neq \mathsf{S}} \text{UNDEF'D}(\mathsf{GpView}_{\mathsf{S}'}())$$

The formula IN-GP_p holds along all cuts, c, such that $p \in \mathsf{GpView}_{\mathsf{S}}(c)$ (when it is defined); OUT-GP_p holds when $p \notin \mathsf{GpView}_{\mathsf{S}}(c)$ (also provided $\mathsf{GpView}_{\mathsf{S}}(c)$ is defined).

GMP-3 (Sequence) All processes exhibit the same sequence of local views, provided the views are defined. Moreover, there is a sequence of cuts along which each local view is a system view:

$$\bigwedge_{0\leq x} \bigwedge_{p} \Diamond \bigwedge_{q}$$
$$\Big(\text{IN-LOCAL}_p^x(q) \Rightarrow \text{DOWN}_q \vee \Big(\mathsf{LocalView}_q() = \mathsf{LocalView}_p() = \mathsf{LocalView}_p^x\Big)\Big) \wedge$$
$$\Big(\Big(\neg\text{IN-LOCAL}_p^x(q) \wedge \bigvee_{y<x} \text{IN-LOCAL}_p^y(q)\Big) \Rightarrow \Box\text{NOTDEF'D}(\mathsf{LocalView}_q^x)\Big).$$

GMP-4 (Liveness) For each event $faulty_p(q)$ (respective, $operating_p(q)$) and each process $p \in \mathsf{GpView}^x$, eventually either p is removed from the group view, or q is removed from it (respective, added to it):

a. $\text{FAULTY}_p(q) \wedge \text{IN-GP}_p \Rightarrow \Big(\Diamond\text{OUT-GP}_q \vee \Diamond\text{OUT-GP}_p\Big)$

b. $\text{OPERATING}_p(q) \wedge \text{IN-GP}_p \Rightarrow \Big(\Diamond\text{IN-GP}_q \vee \Diamond\text{OUT-GP}_p\Big).$

GMP-3 is equivalent to requiring that each local view eventually becomes a group view. The presence and placement of the $\Diamond$ modality forces a group view to exist along some consistent cut in an execution. This too, is why we cannot bind $\mathsf{LocalView}_q$ to a version number. Local views indexed by version numbers are static – the composition of a process's x^{th} local view will never change. If c is the witness cut for $\Diamond$, then omitting the version superscript forces $\mathsf{LocalView}_q(c)$ (for every $q \in \mathsf{LocalView}_p^x$) to be identical to $\mathsf{LocalView}_p^x$ at least along c. Had we included the version number in the equality clause, we would not have been able to conclude that group views necessarily exist, since the local views need not have been identical simultaneously.

Finally since each process executes at least one event between local views x and $x+1$, the corresponding group views will exist along cuts that are related by $\ll$, so it makes sense to talk about the x^{th} group view, which we denote GpView^x.

4 A Protocol Solving Strong GMP

Our solution to Strong GMP, the Strong Group Membership Protocol (hereafter S-GMP), is asymmetric and centralized: a distinguished core member, denoted mgr, coordinates

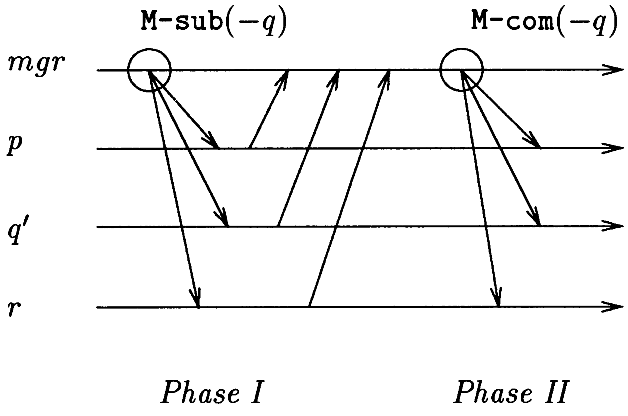


Figure 2: Two-Phase Communication Structure of Simple S-GMP.

updates among all core members' local views. In a symmetric, distributed solution [14, 3] all core members would behave identically and make updates independently. We chose the centralized approach for two reasons: it requires only $O(n)$ point-to-point messages, instead of $O(n^2)$, and it is a simpler paradigm within which to reason. While *mgr* 's failure is more troublesome to handle than an outer (non-*mgr*) member's, the benefits of the centralized approach, coupled with the low probability of the *mgr* failing outweigh these concerns.

An important aspect of S-GMP is the lack of restrictions on changes to a group view. Specifically, there is no upper limit on the number of processes one can add to GpView^x to form GpView^{x+1}; if removing processes from GpView^x, the upper limit is the size of the largest minority subset of GpView^x. This flexibility broadens fault-tolerance, and enables a membership service defined by Strong GMP to adapt quickly to changes in system load. The Appendix contains the complete protocol.

4.1 Simple S-GMP

While we assume in this section that *mgr* does not fail, the protocol we present has a more complicated communication structure and degree of coordination than this assumption warrants. Indeed, if we knew *mgr* could not fail we would already have a single, fault-tolerant process. Anticipating *mgr* 's failure simplifies describing reconfiguration in Section 4.2.

When *mgr* suspects an outer member's (or some subset of outer members'), say r's, failure, it initiates a two-phase update algorithm. In Phase I (Figure 2) *mgr* proposes q's removal by multicasting a *submit* message, M-sub($-q$), to the members of its local view (multicasts are not failure-atomic). *mgr* then waits for each member to respond, or to start believing a member faulty. In this way, at the end of Phase I, all core members that *mgr* does not believe faulty, believe q faulty. If *mgr* receives responses from a majority subset of its current local view, it multicasts a *commit* message, `M-com`$(-q)$, in Phase II;[6] *mgr* must block if it does not receive a majority response. If local views are identical at the beginning of this protocol, because *mgr* is a single process, local views are identical at the end of it.

The submit message coordinates belief among the core in q's faultiness; the commit message tells outer members that the group has reached *agreement* on q's failure and that they should now remove q from their own local views. However, because *mgr* does not receive responses from outer members it believes faulty, it cannot know whether these members received its submit message. From *mgr*'s perspective, these members may not be aware of the current update to the group view, rendering core-wide agreement on the new view *contingent* upon the subsequent removal of these 'faulty' members. The Gossip assumption ensures that operational outer processes become aware of such contingencies.

When adding to the group view, *mgr* sends the new process(es) p a `State-Xfer` message giving p permission to join and informing p of all relevant system state. *mgr* awaits a reply (or suspicion of p's faultiness) and then multicasts the commit message to the entire new group. To simplify bookkeeping, new members begin with local version equal to the group version in which their addition resulted.

4.2 Full S-GMP

When *mgr* is believed to have failed the outer members execute a *reconfiguration algorithm* to select a new coordinator and, if necessary, reestablish the group view. Local view agreement may be lost, for example, when *mgr* fails in the middle of a `M-com()` multicast. In Figure 3 local views differ along the second cut so the group view is undefined.

Reconfiguring successfully involves solving two problems: *succession* – which process(es) should initiate reconfiguration and which should assume the *mgr* role at the end; and *progression* – which update should a reconfiguration initiator propose to resolve core members' inconsistencies?

A reconfigurer must be able to determine the last defined group view and propagate the correct proposal for the succeeding group view. Extrapolating from Figure 3, we see that proposals may also be partially known among the current group view.

The most difficult aspect of reconfiguring involves *invisible commits.* An invisible commit occurs when the only processes receiving a commit message fail, or are believed faulty by the

[6] Typically, a phase of communication consists of a multicast from a single process to a group of processes and their responses back to the initiator. In fact, Simple S-GMP is one-and-one-half phases, but this is awkward.

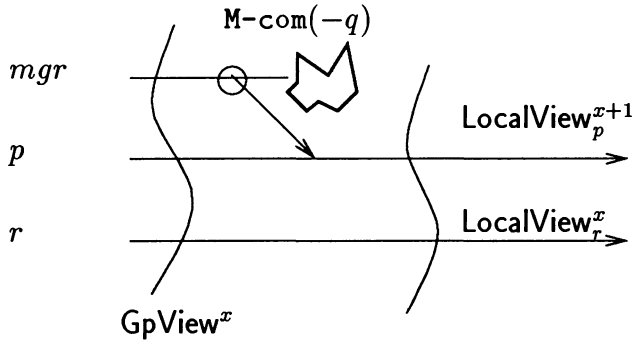


Figure 3: *mgr* 's Failure Results in Undefined Group View.

rest of the group. This is significant for reconfiguration: while no subsequent reconfigurer will ever *know* whether these processes committed the change to their local views, GMP-3 requires that if an invisible commit did occur, the remaining core members must behave consistently. It is imperative, then, that every invisibly committed update be detectable by every reconfigurer. We can ensure this only if all initiators (whether *mgr* or a reconfigurer) attempting to install the x^{th} group view vie for the requisite majority responses from among the same set of processes.

4.2.1 The Reconfiguration Algorithm

Unlike the *mgr*-initiated algorithm, reconfiguration requires three phases in the worst case. This is an outgrowth of the Sequence requirement (GMP-3) and the possibility of invisible commits; we discuss this below in more detail. For simplicity, we present the algorithm here as always using three phases ([17] discusses the cases when two suffice).

For p with $\text{ver}(p) = x-1$, let $\mathsf{NextUpdate}_p$ be the tuple $[< v, x >, \text{rank}(i)]_p$, where v is the value p is waiting to commit to form $\mathsf{LocalView}_p^x$, and $\text{rank}(i)$ is the rank (in $\mathsf{LocalView}_p^{x-1}$) of the initiator that submitted $< v, x >$. $\mathsf{LastCommit}_p$ is the value p committed to form $\mathsf{LocalView}_p^{x-1}$. *state*($p$) is p's local state information: $\text{ver}(p)$, $\mathsf{NextUpdate}_p$, and $\mathsf{LastCommit}_p$.

In the first phase, the initiator r, multicasts a reconfiguration *interrogate* message, `R-int`(*state*(r)), to its local view. The reconfigurer then awaits responses from the outer processes, or its own belief in their faultiness. Upon receiving `R-int`(*state*(r)) a core member that is lagging behind r adopts r's local state as its own (committing the appropriate value, and so forth). Every core member, whether it just updated its local state or not, responds to the reconfigurer with its current local state, *state*().

If a majority respond, then r uses the information it received to determine an update value, say v, and version number, say x, whose execution would result in a new group view. The initiator multicasts this event as the Phase II reconfiguration *submit* message, $\texttt{R-sub}(< v, x >)$. After obtaining a second majority response acknowledging $\texttt{R-sub}(< v, x >)$, r multicasts the Phase III reconfiguration *commit* message, $\texttt{R-com}(< v, x >)$. Again, majority response to $\texttt{R-int}(\mathit{state}(r))$ and $\texttt{R-sub}(< v, x >)$ are essential in maintaining GMP-2 and GMP-3; without either, r must block. If the committed operation is the addition of a set of processes, say Q, then $q \in \mathsf{Q}$ must respond with any pending $\mathsf{NextUpdate}_q$ value it may have (Figure 4). This is necessary to maintain GMP-2 and GMP-3 in cases where Q had already joined at the behest of a previous intiator, for example *mgr*. If *mgr* had been able to propose and additional update to Q, say $\texttt{M-sub}(< +\mathsf{R}, x + 2 >)$, it may also have been able to commit $< +\mathsf{R}, x + 2 >$ invisibly to r and Q.

Definition An update initiator (either *mgr* or a reconfigurer) is *successful* for a submission ($\texttt{M-sub}(< v, x >)$ or $\texttt{R-sub}(< v, x >)$) if a majority subset of the initiator's local view respond to the submission. In this case, we say the submitted value is *stable.* ∎

A successful initiator is able, if it does not fail, to commit the value it submitted. In this light, GMP-3 means that all successful version x initiators must make identical proposals. The local state information collected during reconfiguration Phase I must allow a reconfigurer to determine the correct update proposal unambiguously.

In S-GMP all successful reconfigurers attempting to install (or complete the installation of) the x^{th} group view propagate *mgr*'s proposal if they become aware of it; they propose *mgr*'s removal if they do not. Unfortunately, as Figure 5 makes clear, asynchrony and inopportune failures can result in there being two different proposals for the same instance of the group view. There, reconfigurer r_1 does *not* learn of *mgr*'s proposal, $< v, x >$, and so proposes *mgr*'s removal for version x (as dictated by Procedure *DetermineProposal* in the Appendix). The subsequent reconfigurer r_2 learns of both proposals and must then decide which to propagate. Correctness requires that only one of the two proposals become stable, and that any non-blocking reconfigurer be able to determine which one it is by the end of Phase I (we discuss how a reconfigurer determines this in Section 4.2.3). By propagating the stable submission, a reconfigurer forces the entire group to act consistently with any invisible commits.

4.2.2 Rules of Succession

We solve the succession problem by imposing a deterministic, *linear ranking* on core members based on seniority in the group view – 'older' core members are ranked higher. This is sensible only if group views are unique and agreed-upon. Let rank(p) denote p's rank. Whenever a

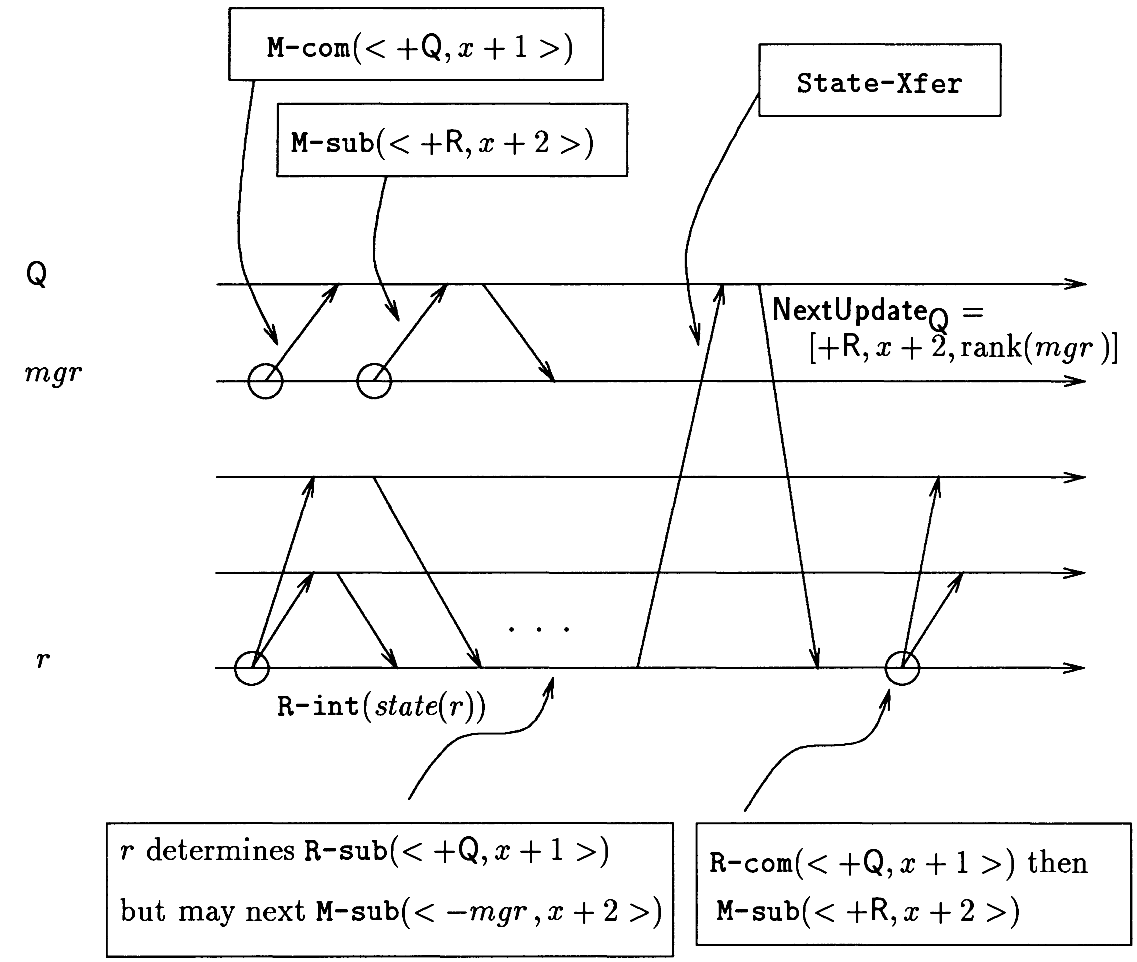


Figure 4: A Situation Requiring New Core Members to Report NextUpdate to Initiator (mgr's commit message cannot reach any processes except Q).

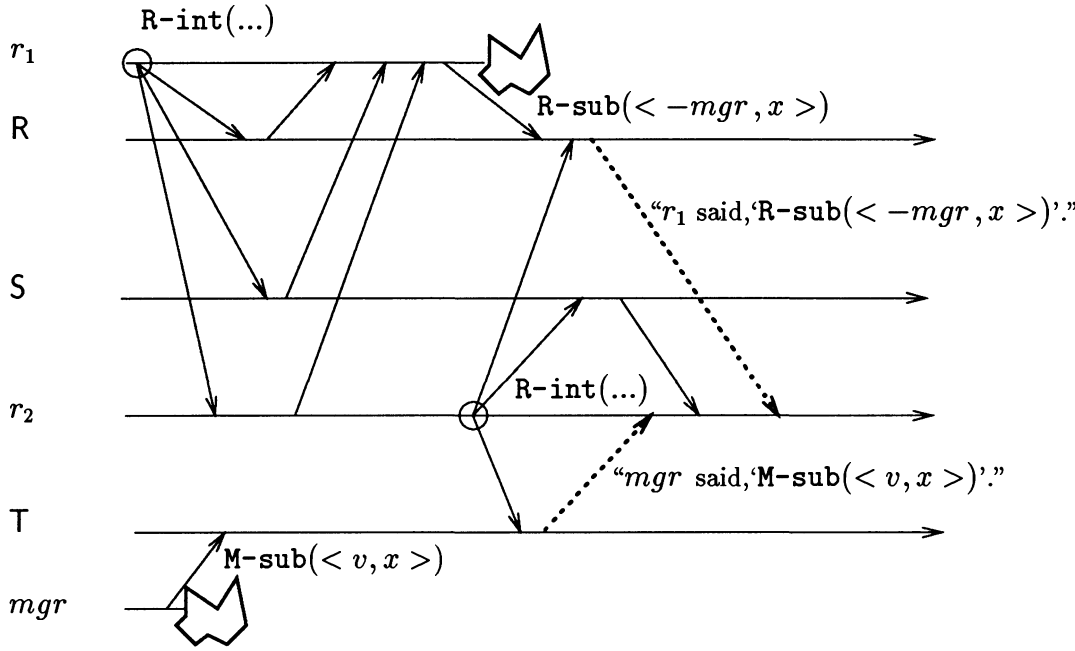


Figure 5: Reconfigurer r_2 Learns of Conflicting Proposals for GpView^x.

process is removed from the group view, the ranks of all higher-ranked processes are decreased by one.

A process initiates reconfiguration when it believes all others ranked higher than itself are faulty. That is, given cut c and $\mathsf{LocalView}_p(c)$,

$$\text{INITIATE}(p) \equiv \bigwedge_{q \in \mathsf{LocalView}_p(c)} \left(\left(\text{rank}(q) > \text{rank}(p) \right) \Rightarrow \text{FAULTY}_p(q) \right)$$

While initiating reconfiguration on $\text{INITIATE}(p)$ can lead to multiple reconfigurations, it guarantees at least one process will undertake reconfiguring. Consider Figure 6 in which $\text{rank}(mgr) = \rho$, $\text{rank}(p) = \rho - 1$, and $\text{rank}(q) = \rho - 2$, and both p and q believe mgr faulty. In the second scenario q expects p, which has crashed, to initiate a reconfiguration; any solution must ensure that q eventually comes to suspect p faulty. In S-GMP, q times-out waiting for p's R-int() message, surmises $\text{FAULTY}_q(p)$, and then initiates reconfiguration. In the third scenario both p and q initiate reconfigurations. S-GMP must also ensure view uniqueness in the face of multiple, concurrent reconfiguration attempts.

Scenario	UP_p	$\text{FAULTY}_q(p)$	$\text{INITIATE}(q)$	$\text{INITIATE}(p)$
1st	True	False	False	True
2nd	False	False	Eventually	False
3rd	True	True	True	True
4th	False	True	True	False

Figure 6: Initiating Reconfiguration: $\text{FAULTY}_p(mgr) \wedge \text{FAULTY}_q(mgr).\text{rank}(mgr) = \text{rank}(p) + 1 = \text{rank}(q) + 2$.

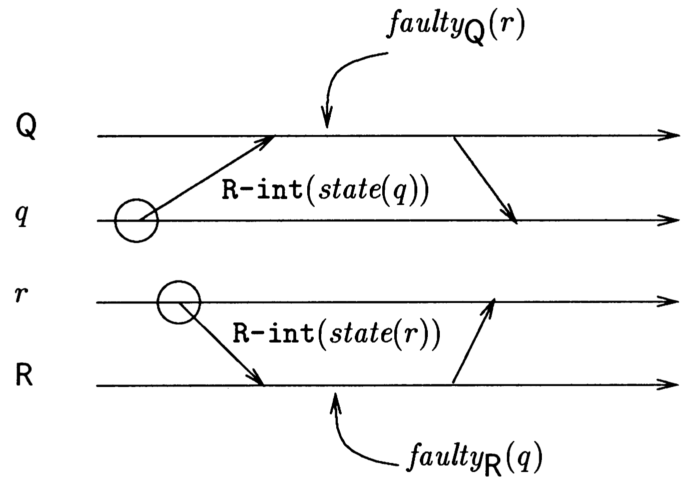


Figure 7: Majority of Responses Needed

4.2.3 Rules of Progression

To understand the difficulties in reconfiguring we examine GMP-2 and GMP-3 more closely. Uniqueness requires that at most one group view exists along any consistent cut. In the situation depicted in Figure 7, Q and R are subsets of GpView^x, and q and r are both initiating reconfiguration. If all members of Q believe r faulty, the Disconnect assumption means they will receive none of r's messages. Analogous statements hold for the members of R regarding q. If r's proposal differs from q's then the members of R will commit a different value than the members of Q. If $\mathsf{R} \cup \{r\}$ eventually remove all of $\mathsf{Q} \cup \{q\}$, and $\mathsf{Q} \cup \{q\}$ eventually remove all of $\mathsf{R} \cup \{r\}$, two distinct group views will exist.

Naively, it would appear that the majority requirement suffices to ensure Uniqueness. However, as Figure 3 makes clear, initiators that may end up installing (submitting and

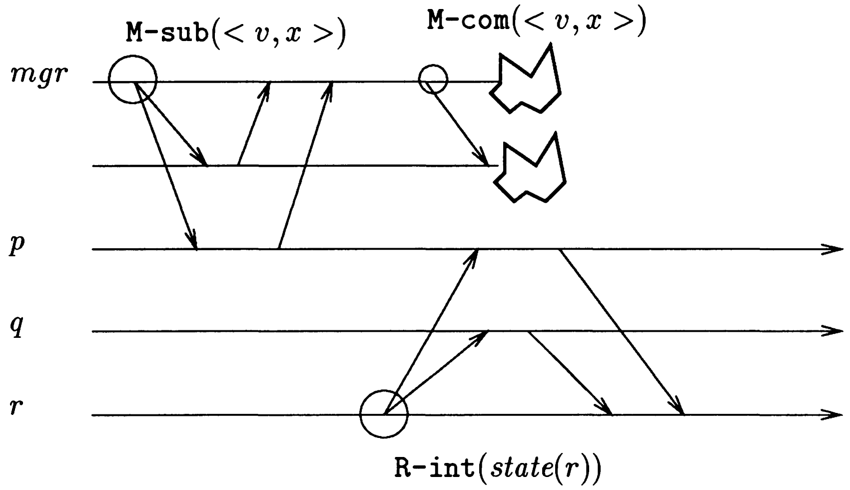


Figure 8: Value $< v, x >$ Committed Invisibly to p, q, and r

committing) the same group version need not begin reconfiguration with identical local views and so may be seeking majority approval from different sets of processes.

Reconfiguration Phase I Responses

Outer processes' responses to R-int($state(r)$) must allow r to determine the nature and composition of all local view inconsistencies, including inconsistencies involving core members that did not respond to r. Local view information alone is insufficient to satisfy GMP-3 (Sequence) as invisible commits are not detectable.

In Figure 8, $< v, x >$ is committed invisibly to p, q, and r. Since all three have identical local views, r will not detect the actual discrepancy. However, p is aware of mgr's intention to commit $< v, x >$, and p can envision a situation in which mgr succeeded in doing so and then failed (in this case, the situation that actually occurred). If p were to forward mgr's intention to commit $< v, x >$, r would then envision the same situation and propagate $< v, x >$ as its Phase II submission. Thus, in addition to its local view, an outer member must also report how it expects to change its local view next.

We have described how a reconfigurer may discover two different values were proposed for the same group version. In S-GMP the reconfigurer propagates the value proposed by the process of *least* rank among those making proposals (Procedure *GetStableProposal* in the Appendix). Proposition 5.7 proves this choices ensures GMP-2 and GMP-3.

4.2.4 Membership Service Startup

Our approach depends on the initial group view being unique, and this is difficult to guarantee in asynchronous systems. We use a heuristic borrowed from previous versions of the Isis system. Briefly, "cold-start" of the MRM is limited to a small, known set of sites. To cold-start, a process first queries these locations to determine whether any others have begun the cold-start procedure. It continues the cold-start procedure only if it determines no others have begun, or if it "outranks" all processes that have concurrently begun cold-starting. Because we iterate this procedure the probability that two cold-starting processes remain unaware of each other diminishes with each round. After a suitable number of successful rounds a process determines it should start the MRM. Although probabilistic, we find this scheme highly successful in practice.[7]

5 Correctness

The proof that S-GMP correctly solves Strong GMP is inductive. In this section we present the more interesting theorems of the inductive step. We show that if GpView^{x-1} is uniquely defined, S-GMP results in exactly one value being committed among the members of GpView^{x-1} to obtain GpView^{x}. That S-GMP satisfies GMP-2 and GMP-3 follows from there.

The major steps in the proof are, first, showing that all initiators attempting to install GpView^{x} do so starting from $\mathsf{LocalView}_p^{x-1}$. As a result, all such initiators compete for majority approval from the same set of processes. We use this result when we show that a reconfigurer knows which of the two proposals it may learn of could not have been stable. While the other proposal may not, in actuality, be stable, by choosing to propagate it, the reconfigurer *cannot* possibly act inconsistently with the subset of the core that is 'invisible' to it. Stated another way, we show that all *successful* initiators propose the same value for GpView^{x}, and that this value is the only one that can possibly be committed.

For brevity, we do not prove all propositions; the full proof of correctness is in [17].

5.1 The Inductive Step

As in Section 4.2.1, $\mathsf{NextUpdate}_p$ is the tuple $[< v, \mathrm{ver}(p) + 1 >, \mathrm{rank}(i)]_p$. For each p, Gossip, Disconnect and INITIATE() mean that $\mathsf{NextUpdate}_p$ is always the proposal of the *lowest-ranked* initiator from which p received proposals for version $\mathrm{ver}(p) + 1$.

[7] In several years of wide use no problems have ever been traced to the restart scheme. Note also that limiting cold-start to a single, known, site suffices to guarantee uniqueness of the initial view but unfortunately this scheme is now vulnerable to a liveness problem: we may be unable to restart the system after a crash.

For process r multicasting message m, $\mathsf{Acks}(r, m)$ is the set of processes from which r receives a message acknowledging, or in response to m. Let

$$\mathsf{Ahead}_r \stackrel{\text{def}}{=} \Big\{p \mid p \in \mathsf{Acks}(r, \texttt{R-int}(state(r))) \wedge \big(\text{ver}(p) > \text{ver}(r)\big)\Big\}$$

Proposition 5.1 If r is a reconfiguration initiator with $\text{ver}(r) = x - 1$, then for every p responding to $\texttt{R-int}(x-1)$, $x - 1 \leq \text{ver}(p) \leq x$. ∎

For process p, let $\mathsf{Faulty}_p = \Big\{q \mid \text{IN-LOCAL}_p(q) \wedge \text{FAULTY}_p(q)\Big\}$.[8] We say GpView^{x-1} is *p-defined* along cut c if p knows at c that every process in $\mathsf{LocalView}_p(c) - \mathsf{Faulty}_p$ has defined its $(x-1)^{st}$ local view. Of course GpView^{x-1} may not be defined globally, but from p's point of view, GpView^{x-1} is (or has been) defined. For a reconfigurer r, GpView^{x-1} is r-defined at the end of Reconfiguration Phase I if every process in $\mathsf{Acks}(r, \texttt{R-int}(x-1)) - \mathsf{Faulty}_r$ reported a local version at least as large as $x - 1$.

Proposition 5.2 Let r be a reconfiguration initiator. Then r proposes version x if and only if GpView^{x-1} is the most recent (i.e. highest-numbered) r-defined system view at the end of Reconfiguration Phase I.

Proof Follows from analyzing procedure *DetermineProposal* in Section 7.
∎

From Proposition 5.2 we infer that an initiator attempting to install version x has local version either $x - 1$ or x. We now show it can only have local version $x - 1$.

Proposition 5.3 For any initiator, r, if r proposes $< v, x >$, then $\text{ver}(r) = x - 1$.

Proof The proof is trivial when $r = mgr$, so suppose r is a reconfiguration initiator, with $\text{ver}(r) \geq x$. When r multicasts $\texttt{R-int}(state(r))$, any process p lagging behind r adopts r's local state as its own.[9] Thus, when it responds to r's interrogate message, $state(p) = state(r)$ making $\text{ver}(r)$ the most recent r-defined version. From Proposition 5.2 r would then propose some value for version $\text{ver}(r) + 1$, and not version x. On the other hand, $\text{ver}(r) < x - 1$ is impossible if GpView^{x-1} is the most recent r-defined view.
∎

Hereafter, we use $\texttt{sub}()$ and $\texttt{com}()$ to denote generic submit and commit messages irrespective of the initiator's role (mgr or reconfigurer).

[8] Faulty_p is implicitly indexical.

[9] It turns out that any process r does not believe faulty at the end of Phase I will have local version at least $\text{ver}(r) - 1$ when it receives $\texttt{R-int}(state(r))$.

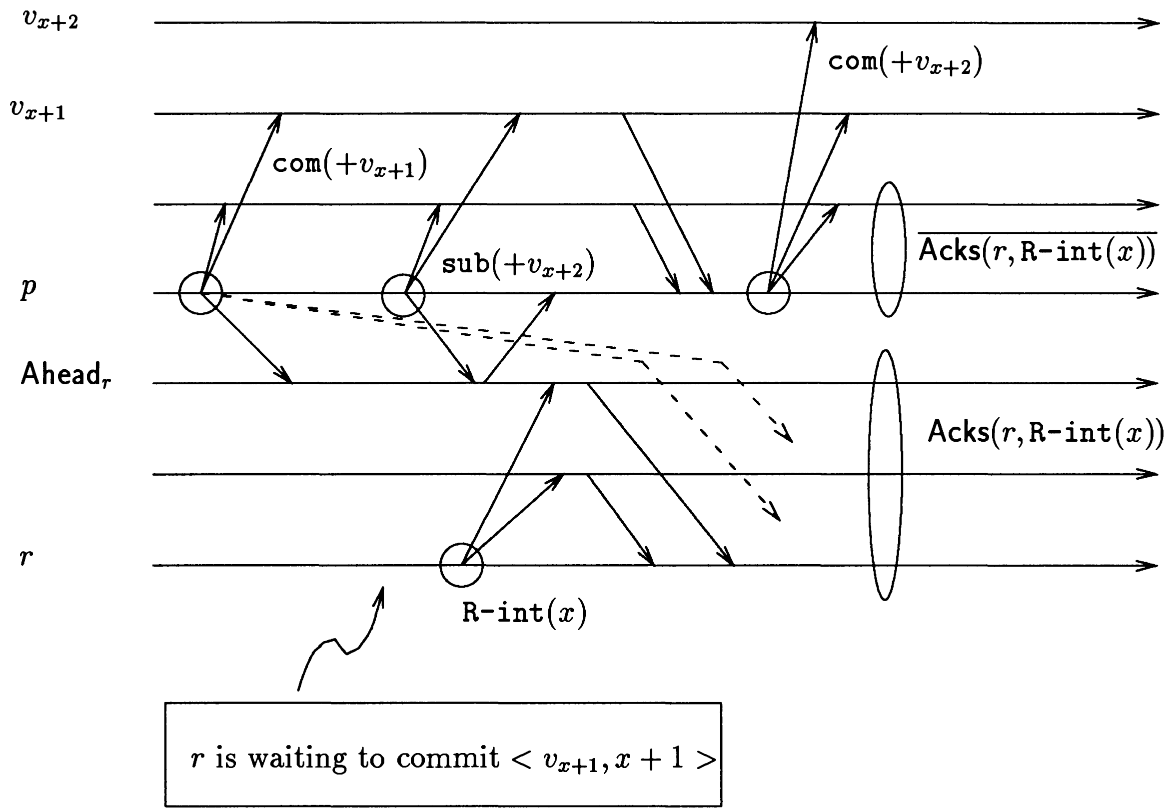


Figure 9: Possible divergence: r is a potentially successful reconfigurer, and p is an initiator that could commit $< v_{x+3}, x+3 >$ without r learning the value v_{x+3}.

To illustrate the difficulty in proving Sequence (GMP-3) consider the following situation, depicted in Figure 9. Let r be a reconfigurer with $\text{ver}(r) = x$, and let $\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))$ (we use $\mathtt{R\text{-}int}(\text{ver}(r))$ rather than $\mathtt{R\text{-}int}(\mathit{state}(r))$ to get explicit reference to r's local version) be a majority subset of $\mathsf{LocalView}_r^x$. Proposition 5.1 means the largest version number observed among r's respondents is $x+1$, so suppose Ahead_r is non-null and let p be a process from which some member of Ahead_r received $\mathtt{com}(< v_{x+1}, x+1 >)$. Suppose further that p also proposed a value, $< v_{x+2}, x+2 >$, for version $x+2$ to which every member of Ahead_r responded. Making matters worse, r can imagine all the processes that did not respond to its own $\mathtt{R\text{-}int}(\mathit{state}(r))$ message may have responded to p's $\mathtt{sub}(< v_{x+2}, x+2 >)$. It may then be the case that $\mathsf{Ahead}_r \cup \overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(\mathit{state}(r)))}$ (and v_{x+1}, if it is an *add*()) form a majority subset of GpView^{x+1}, thereby allowing p to commit view $x+2$. Trouble arises if $\overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(\mathit{state}(r)))}$ (and v_{x+1} and v_{x+2}, if both are *add*() operations) is a majority subset of GpView^{x+2}; neither r nor any process in $\mathsf{Acks}(r, \mathtt{R\text{-}int}(\mathit{state}(r)))$ can know what value p would propose for view $x+3$.

Proposition 5.4 shows that when r is successful and $< v_{x+1}, x+1 >$ is a *remove*() operation, no previous initiator (like p) can commit a version greater than $x+1$. Proposition 5.5 shows that when $< v_{x+1}, x+1 >$ is an *add*() operation, it is possible for p to continue committing new group views and for r to lag behind p. However, if both are successful for a given group version, both commit the same value. These propositions address exactly the situation when it appears S-GMP could violate GMP-2 and GMP-3: when two initiators are successful for the same group version. Propositions 5.4 and 5.5 prove that even when initiators do not vie for majorities from among the same set of core members (their local views differ), S-GMP is safe.

Proposition 5.4 Let r be a reconfiguration initiator with $\text{ver}(r) = x$. Let $\mathsf{Ahead}_r \subseteq \mathsf{Acks}(r, \mathtt{R\text{-}int}(x))$ report local version $x+1$, and let p be a process from which some member of Ahead_r received $\mathtt{com}(< v_{x+1}, x+1 >)$. If $\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))$ is a majority subset of $\mathsf{LocalView}_r^x$ and $< v_{x+1}, x+1 >$ is the removal of a set of processes from GpView^x, then p cannot be successful for any view numbered higher than $x+1$.

Proof Let $q \in \mathsf{Ahead}_r$ and let p be as described. Since q received $\mathtt{R\text{-}int}(x)$ from r after $\mathtt{com}(< v_{x+1}, x+1 >)$ from p, it must be that $\text{rank}(r) < \text{rank}(p)$, so $\text{FAULTY}_q(p)$ holds for every such q in $\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))$ (by Gossip). As a result, the initiator p can be successful for $x+2$ if and only if $\overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))}$ is a majority subset of $\mathsf{LocalView}_p^{x+1} = \mathsf{LocalView}_p^x - v_{x+1}$.

Observe that $\mathsf{LocalView}_r^x = \mathsf{Acks}(r, \mathtt{R\text{-}int}(x)) \cup \overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))}$, and that r cannot have received $\mathtt{ack}(\mathtt{R\text{-}int}(x))$ responses from the members of v_{x+1};[10] in other words, $v_{x+1} \subseteq$

[10]Reconfigurer r must have received p's proposal to remove v_{x+1} or else q would not have received r's

$\overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))}$. Thus p can commit $\mathsf{LocalView}_p^{x+2}$ if and only if $\left(\overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))} - v_{x+1}\right)$ is a majority of $\mathsf{LocalView}_p^{x+1}$. Let $\alpha = \mid \mathsf{Acks}(r, \mathtt{R\text{-}int}(x)) \mid$ and $\bar{\alpha} = \mid \overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))} \mid$. Then initiator p is successful for $< v_{x+2}, x+2 >$ if and only if

$$\frac{\bar{\alpha} - \mid v_{x+1} \mid}{\mid \mathsf{LocalView}_p^{x+1} \mid} = \frac{\bar{\alpha} - \mid v_{x+1} \mid}{\mid \mathsf{GpView}^x \mid - \mid v_{x+1} \mid} = \frac{\bar{\alpha} - \mid v_{x+1} \mid}{\alpha + \bar{\alpha} - \mid v_{x+1} \mid} > \frac{1}{2} \Leftrightarrow \bar{\alpha} - \mid v_{x+1} \mid > \alpha$$

contradicting the assumption that $\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))$ is a majority subset of GpView^x.

∎

Proposition 5.5 Let r be a reconfiguration initiator with $\mathrm{ver}(r) = x$. Let Ahead_r be non-null and let p be a process that sent $\mathtt{com}(< v_{x+1}, x+1 >)$ to some member of Ahead_r. Then if r is successful for $< v_{x+1}, x+1 >$, then if p later submits $< v_{x+2}, x+2 >$, either r or p, but not both, can be successful for version $x+2$.

Proof GMP-3 will be violated if p is able to commit $< v_{x+2}, x+2 >$ and r is able to commit $< -\overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))}, x+2 >$. We proceed by analyzing the messages arriving at v_{x+1}.

(a) Consider Figure 10 (top diagram). The two-headed split-arrow message from r to v_{x+1} represents the two possibilities for the arrival of r's commit message,

$$m = \mathtt{R\text{-}com}(< v_{x+1}, x+1 >) : \mathtt{M\text{-}sub}(< -\overline{\mathsf{Acks}(r, \mathtt{R\text{-}int}(x))}, x+2 >),$$

at v_{x+1}. p's commit message, $\mathtt{com}(< v_{x+1}, x+1 >)$, to v_{x+1} is a dashed because of the possibility that it may not be received. We elide r's Phase II submit message.

Suppose the members of v_{x+1} receive m from r before they receive $\mathtt{com}(< v_{x+1}, x+1 >)$ from p. Since r's message gossips its belief in p's faultiness, the members of v_{x+1} will never receive another message from p. In particular the members of v_{x+1} will not receive p's subsequent $\mathtt{M\text{-}sub}(< v_{x+2}, x+2 >)$. We say r *owns* v_{x+1}.

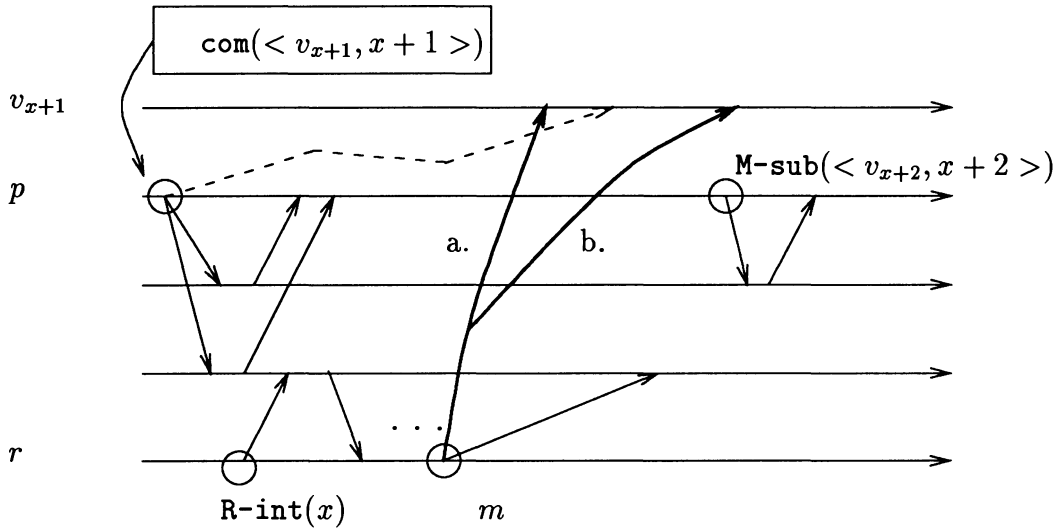
Using α and $\bar{\alpha}$ as defined in Proposition 5.4, p is successful for version $x+2$ if and only if $\bar{\alpha} > \alpha + \mid v_{x+1} \mid$ and r is successful for version $x+2$ if and only if $\alpha + \mid v_{x+1} \mid > \bar{\alpha}$. Both conditions cannot hold.

(b) If the processes in v_{x+1} receive m from r after $\mathtt{com}(< v_{x+1}, x+1 >)$ and before $\mathtt{M\text{-}sub}(< v_{x+2}, x+2 >)$ from p, the analysis is the same as in (a); r owns v_{x+1} once the members of that set receive m.

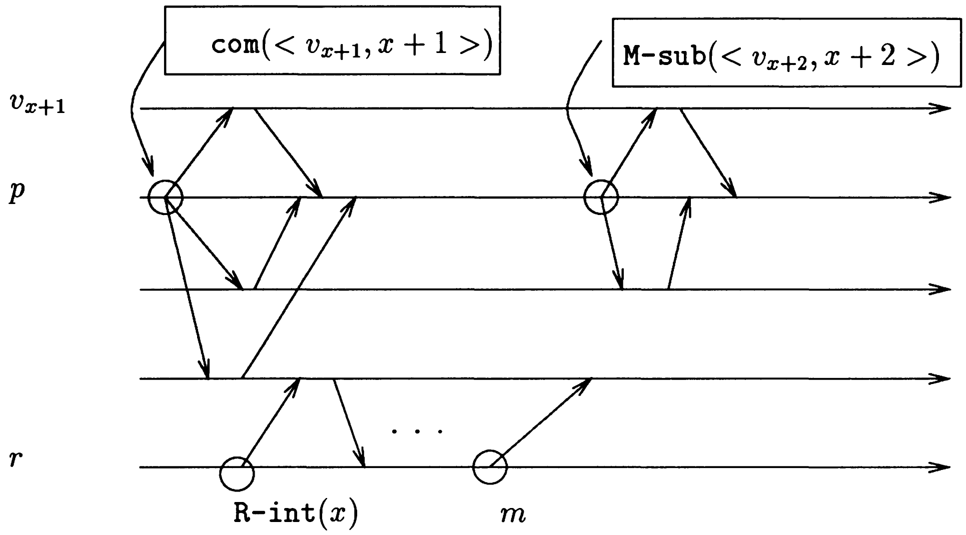
(c) In the last case (Figure 10 bottom), p owns v_{x+1} if its $\mathtt{M\text{-}sub}(< v_{x+2}, x+2 >)$ message, gossiping p's belief in r's faultiness, arrives at v_{x+1} before r's $\mathtt{R\text{-}com}()$ message does. Then

---

$\mathtt{R\text{-}int}(x)$; had r not received and responded to p's proposal, p's commit message to q would have gossiped $\mathit{faulty}_p(r)$.



Cases (a) and (b): r owns v_{x+1}.



Case (c): p owns v_{x+1}.

Figure 10: Case Analysis for Proposition 5.5

p is successful for version $x+2$ if and only if $\bar{\alpha} + \mid v_{x+1} \mid > \alpha$ and r is successful for version $x+2$ if and only if $\alpha > \bar{\alpha} + \mid v_{x+1} \mid$. Again, both conditions cannot hold. ∎

It remains to prove that when r learns of two version-identical proposals (how this situation may arise was described in Section 4, Figure 5), the proposal submitted by the initiator of least rank is the only one that could have been invisibly committed. That is, r correctly identifies the initiator and proposal that could have been successful; as a result r cannot act inconsistently with any invisible commits. Referring to the S-GMP algorithm in the Appendix this necessity arises in determining $v2$ when $\mathsf{Ahead}_r \neq \emptyset$, and in determining $v1$ when $\mathsf{Ahead}_r = \emptyset$.

Let GpView^{x-1} is the most recent r-defined group view and define $\mathsf{Submissions}_r(x)$ to be the set of proposed next updates for version x that r learns about in response to its `R-int()` message: for some initiator i,

$$\mathsf{Submissions}_r(x) = \Big\{ v_x \mid \exists p \in \mathsf{Acks}(r, \mathtt{R\text{-}int}()) : \mathsf{NextUpdate}_p = [v_x, x, \mathrm{rank}(i)] \Big\}$$

We first describe the composition of $\mathsf{Submissions}_r(x)$, showing that every reconfigurer proposing version x either propagates *mgr*'s proposal for version x or proposes *mgr*'s removal.

Proposition 5.6 For all versions x, $\mid \mathsf{Submissions}_r(x) \mid \leq 2$.

Proof Inspecting procedure *DetermineProposal*, different submissions for the same view can arise only from *mgr* and from a reconfiguration initiator proposing *mgr*'s removal. The latter occurs if and only if the initiator did not learn of any outstanding proposal made by *mgr*; that is if $\mathsf{Submissions}_r(x) = \emptyset$.

∎

We say $\mathsf{Submissions}_r(x)$ is *bivalent* if it contains two distinct values. Corollary 5.1 follows by examing procedure *DetermineProposal* in the Appendix. It shows that all reconfigurers either propagate *mgr*'s unique submission for view x or propose *mgr*'s removal.

Corollary 5.1 Let r and r' be reconfigurers proposing version x. Then if both their $\mathsf{Submissions}(x)$ sets are bivalent, they are identical:

$$\Big(\mid \mathsf{Submissions}_r(x) \mid \;=\; \mid \mathsf{Submissions}_{r'}(x) \mid \;=\; 2\Big) \implies \mathsf{Submissions}_r(x) = \mathsf{Submissions}_{r'}(x).$$

∎

With these preliminaries we can now prove only one of these two proposals could possibly have been committed (invisibly or otherwise), and that all reconfigurers can distinguish which of the two it was. This proposition is vital to the inductive step: it shows that in going from GpView^{x-1} to GpView^{x} one and only one value can be committed by any member of GpView^{x-1} as the same value is proposed by any successful initiator for the x^{th} group version.

Proposition 5.7 Let r be a reconfiguration initiator. If $\mathsf{Acks}(r, \texttt{R-int}(\mathit{state}(r)))$ is a majority subset of $\mathsf{LocalView}_r$ and $\mathsf{Submissions}_r(x)$ is bivalent, then r can distinguish which of the two values proposed could *not* have been committed invisibly.

Proof Let r be as described, and let $\mathsf{Submissions}_r(x) = \{< v, x >, < v', x >\}$. Let p be the process of least rank among those reported to have submitted $< v, x >$, and let p' be the process of least rank among those reported to have submitted $< v', x >$. r must decide which of the two, p or p', could not have been successful for version x. We show that r chooses correctly when it is the *first* bivalent reconfigurer for version x, then prove the proposition inductively.

In order for either value to have been committed, its initiator must have garnered majority approval from its local view for the submitted value. Since both p and p' make version x submissions, both must have local version $x - 1$ (Proposition 5.3). Without loss of generality assume $\text{rank}(p) < \text{rank}(p')$, and consider the possible roles p could have had:

a) If p were the *mgr*, its proposal $\texttt{M-sub}(< v, x >)$ could not have reached a majority subset of GpView^{x-1}; if it had, then p' would have learned of it from some process in $\mathsf{Acks}(p', \texttt{R-int}(x - 1))$. Since r is the first bivalent reconfigurer, $\mathsf{Submissions}_{p'}(x)$ would have to be the singleton $\{< v, x >\}$, which p' would have propagated in *Determine-Proposal.*

Thus, $< v, x >$ is not stable because p cannot have been successful for $< v, x >$. Looking at *DetermineProposal*, initiator r propagates $< v', x >$ because it was submitted by p', the initiator with least rank among those mentioned in $\mathsf{Submissions}_r(x)$.

b) If $p \neq mgr$, it is successful for $< v, x >$ if and only if $\mathsf{Acks}(p, \texttt{R-sub}(< v, x >))$ is a majority subset of GpView^{x-1}. Both p and p' were able to make proposals so their first response sets were majority subsets. Let A be their intersection:

$$\mathsf{A} = \mathsf{Acks}(p, \texttt{R-int}(x - 1)) \bigcap \mathsf{Acks}(p', \texttt{R-int}(x - 1)).$$

The gossip property and $\text{rank}(p) < \text{rank}(p')$ mean that a majority of p's local view believe it faulty upon receiving $\texttt{R-int}(x - 1)$ from p'. Disconnect means that

$$recv_a(p, \texttt{R-int}(x-1)) \rightarrow recv_a(p', \texttt{R-int}(x-1)) \rightarrow faulty_a(p) \qquad \forall a \in \mathsf{A}.$$

The question is whether any $a \in \mathsf{A}$ receives $\texttt{R-sub}(< v, x >)$ from p, which could only happen before it receives $\texttt{R-int}(x-1)$ from p':

$$recv_a(p, \texttt{R-sub}(< v, x >)) \rightarrow recv_a(p', \texttt{R-int}(x-1)) \rightarrow faulty_a(p).$$

Now, any such a would have forwarded $< v, x >$ as part of $\mathsf{NextUpdate}_a$ to p' when responding to $\texttt{R-int}(x-1)$, in which case either

1. $\mathsf{Submissions}_{p'}(x)$ is bivalent, violating the assumption that r is the first bivalent reconfigurer, or
2. every process in $\mathsf{Acks}(p', \texttt{R-int}(x-1))$ reported $< v, x >$ as its next pending update. But in this case, $\mathsf{Submissions}_{p'}(x)$ would again be the singleton $\{< v, x >\}$, which p' would have propagated in *DetermineProposal*.

Thus, no $a \in \mathsf{A}$ received $\texttt{R-sub}(< v, x >)$ from p, meaning the only processes that could have are those in $\mathsf{Acks}(p, \texttt{R-int}(x-1)) - \mathsf{A}$. This cannot be a majority subset of GpView^{x-1} since it is disjoint from $\mathsf{Acks}(p', \texttt{R-int}(x-1))$ which is a majority subset.

We have just proven the base case for the proposition – when r is the first bivalent reconfigurer (it is not hard to see that 'first' is meaningful and well-defined in the context of successful initiators for a given group view). If r's proposal reaches a majority subset of GpView^{x-1} then the value it propagated will be chosen by the next reconfigurer to get a majority response to its Reconfiguration Phase I interrogate message as r would be the submitter with least rank. If r's proposal does not Πreach a majority subset, the next bivalent reconfigurer will nonetheless choose as r did and so propagate the correct value.

∎

Corollary 5.2 If GpView^{x-1} is defined, there is at most one stably-defined proposal for group version x.

Proof Proposition 5.7 proves that *GetStableProposal* correctly chooses the only proposal for a given group view that *could* have been committed invisibly to a reconfiguration initiator when its Phase I response set is bivalent. When the set is univalent or empty, it is not hard to see that *DetermineProposal* is safe. If this initiator reaches its commit stage, its proposal is stably-defined and identical to the other stably defined proposals for version x.

∎

Theorem 5.1 (Identical Local Views) If GpView^{x-1} is defined, then all members that survive to define local version x have identical local x^{th} views.

Proof The result follows from Corollary 5.2; no process commits a local view for version x that differs from any other processes' version x because all proposals that can possibly reach the commit stage are identical.

∎

Note that Theorem 5.1 implies no temporal constraints on local views, merely that if p ever defines an x^{th} local view, and if q, too, ever defines an x^{th} local view, then these two are identical. It does not require $\mathsf{LocalView}_p^x$ and $\mathsf{LocalView}_q^x$ to exist together in some global state. Thus, to prove S-GMP satisfies GMP-3 requires slightly more work.

5.2 Message Complexity

[16] proves S-GMP (with two minor modifications) is message minimal for Strong GMP. Moreover, S-GMP is also phase-minimal. The message-minimality proof gives the required direction of information flow as well as the content of each message. In S-GMP the pattern of required communication is arranged to minimize the length of the message-path from the beginning of the update algorithm to the end. For example, Figure 11 shows two ways to organize the distributed event "send a message to every process in S and collect responses or time-out".

Observe that the Phase I submit message is unnecessary if *mgr* knows a majority of the non-faulty outer processes already believe a process, say q, faulty. In this light, a contingent update, piggy-backed on a commit message, can serve as the submit message for the next view change. We can thus compress successive instances of Simple S-GMP if *mgr* makes known when it multicasts the commit message, exactly how it plans to change the group view next. In Figure 12, process q' crashes before responding to `M-sub`$(-q)$, causing *mgr* to suspect q' faulty. By appending `M-sub`$(-q')$ to `M-com`$(-q)$, *mgr* indicates that it wants to remove q from the just-formed group view. Outer processes respond to the piggy-backed commit-submit message as they would respond to a plain submit message. The correctness proofs (Propositions 5.5 and 5.7 in the previous section require only slight modifications to handle this optimization).

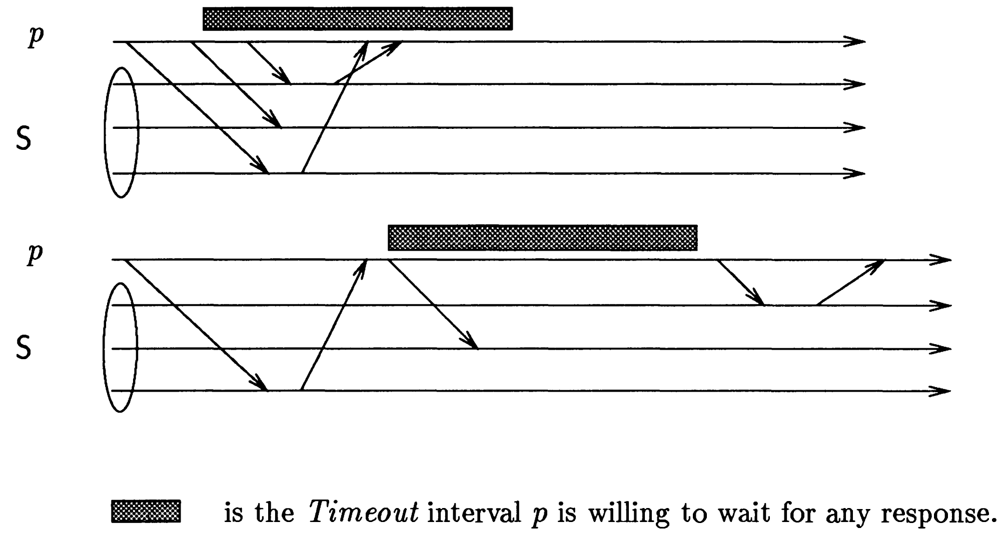


Figure 11: Two possible communication patterns accomplishing "send a message to every process in S and collect responses or time-out."

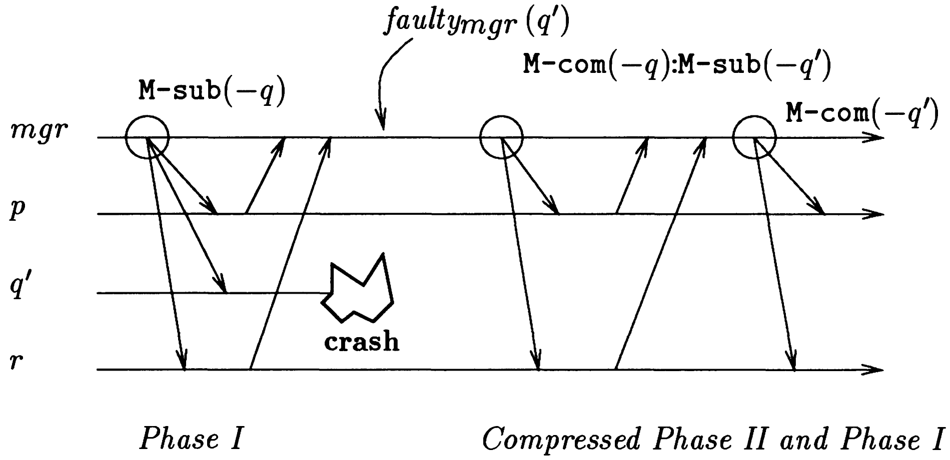


Figure 12: Compressing Successive Instances of Simple S-GMP.

When we can take advantage of compressing phases we gain substantially. Define $n_x \stackrel{\text{def}}{=} \mid \mathsf{GpView}^x \mid$, and let a_x be the number of processes added to GpView^x and r_x be the number of processes removed from GpView^x. Then Y successive compressed updates (with no reconfiguring) requires and initial n_x submit messages, $n_x - r_x$ acknowledgement messages, a handshake of $2a_x$ `State-Xfer` and `ack(State-Xfer)` messages, $n_x - r_x + a_x$ commit messages. To update the new GpView^{x+1}, there are $n_x - r_x + a_x - r_{x+1}$ messages to acknowledge `M-sub`$(< v_{x+1}, x+1 >)$, followed again by $2a_{x+1}$ messages for the `State-Xfer-ack(State-Xfer)` handshake, and $n_x - r_x + a_x - r_{x+1} + a_{x+1}$ commit messages...

$$n_x + \sum_{k=x}^{x+Y}\left(\left(n_x - \sum_{i=x}^{k} r_i + \sum_{j=x}^{k-1} a_j\right) + 2a_k + \left(n_x - \sum_{i=x}^{k} r_i + \sum_{j=x}^{k} a_j\right)\right) =$$

$$(2Y+1)n_x + \sum_{k=x}^{x+Y} a_k + 2\sum_{k=x}^{x+Y}(x+Y-k)\delta_k$$

where $\delta_k = a_k - r_k$. When we cannot take advantage of piggy-backing, there are

$$Yn_x + \sum_{k=x}^{x+Y}(x+Y-k)\delta_k$$

additional messages.

6 Conclusion

We have described an approach to the asynchronous system membership problem which provides very strong distributed consistency guarantees, and yet is inexpensive in comparison even to less powerful membership services. Current distributed systems lack membership services, forcing application designers to solve this problem repeatedly through ad-hoc, and often inconsistent, mechanisms. As technology such as GMP becomes more widely available, we believe that a major obstacle to reliable distributed software development will have been removed.

References

[1] A. El Abbadi and S. Toueg. Maintaining Availability in Partitioned Replicated Databases. *ACM Transactions on Database Systems*, 14(2):264–290, June 1989.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms in Broadcast Domains. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth Workshop on Distributed Algorithms and Graphs; Israel.* Springer-Verlag, 1992. Lecture Notes in Computer Science.

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

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed systems. In *Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

[6] K. P. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[7] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *A.C.M. Transactions on Computer Systems*, 3(1):63–75, 1985.

[8] B. A. Coan and G. Thomas. Agreeing on a Leader in Real-Time. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 166–172, December 1990.

[9] F. Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. Technical Report RJ 5964, IBM Almaden Research Center, August 1990. Revised from March, 1988.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[11] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the A.C.M.*, 21(7):558–565, 1978.

[12] K. Marzullo, K. Birman, R. Cooper, and M. Wood. Tools for Distributed Application Management. Technical Report TR 90-1136, Cornell University, June 1990.

[13] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership Algorithms for Asynchronous Distributed Systems. In *Proceedings of the IEEE 11th International Conference On Distributed Computing Systems*, May 1991.

[14] S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol Based on Partial Order. In *Proceedings of the IEEE International Working Conf on Dependable Computing for Critical Applications*, February 1991.

[15] R. F. Rashid. Threads of a New System. *Unix Review*, 4:37–49, August 1986.

[16] A. M. Ricciardi. Practical Utility of Knowledge-Based Analyses : Optimizations and Optimality for and Implementation of Asynchronous Fail-Stop Processes. In *Fourth Conference on the Theoretical Aspects of Reasoning About Knowlege.* Morgan Kaufmann, March 22-25 1992.

[17] A. M. Ricciardi. *The Asynchronous Membership Problem.* PhD thesis, Cornell University, January 1993.

[18] A. M. Ricciardi, K. P. Birman, and P. Stephenson. The Cost of Order in Asynchronous Systems. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth Workshop on Distributed Algorithms and Graphs; Israel.* Springer-Verlag, 1992. Lecture Notes in Computer Science.

[19] R. D. Schlichting and F. B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM TOCS*, 1(3):222–238, August 1983.

[20] F. B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM TOCS*, 2(2):145–154, May 1984.

[21] M. D. Skeen. *Crash Recovery in a Distributed Database System.* PhD thesis, University of California at Berkeley, May 1982.

7 Appendix: The S-GMP Algorithm

We abbreviate "either add or remove Q" with $\pm$Q. If Q is a set or process identifiers, $Mcast_p(\mathsf{Q}, m)$ denotes the compound action $\forall q \in \mathsf{Q} : (send_p(q, m))$. $Mcast_p(\mathsf{Q}, m)$ is an indivisible action only in the sense that p does not execute any other events until all messages are sent; it is not failure-atomic. The message $\mathtt{ack}(m)$ acknowledges receipt of message m. We do not explicitly show gossiping, or channel-disconnect, but assume these are done transparently.

Task : *mgr*

```
while (true)
    repeat
        GetUpdate(v1);
    until (v1 ≠ nil-id);
    Mcast_mgr (LocalView_mgr , M-sub(±v1));

    while (v1 ≠ nil-id) /* Compressed algorithm loop. */
        forall p ∈ LocalView_mgr
            await either recv_mgr (p, ack(M-sub(±v1))) or faulty_mgr (p);
        if (majority of LocalView_mgr didn't respond)
            crash_mgr ;
        DoCommit(v1, ±); /* Update LocalView_mgr according to ±. */
        GetUpdate(v2);
        if (Joining new members)
            Mcast_mgr (v1, Join : State-Xfer);
            forall p' ∈ v1
                await either recv_mgr (p', ack(Join) : NextUpdate_p') or faulty_mgr (p');
            if (NextUpdate_v1 ≠ ⊥)
                v2 ← NextUpdate_v1;
        Mcast_mgr (LocalView_mgr , M-com(±v1) : M-sub(±v2));
        v1 ← v2;
/* end mgr Task */
```

Task: Outer Processes, p

$recv_p$(mgr ,`M-sub`($\pm v1$));
DoPreCommit($v1$, $\pm$); /* Mark $v1$ faulty or operational. */
repeat
    $send_p$(mgr , ack(`M-sub`($\pm v1$)));
    await either $recv_p$(mgr ,`M-com`($\pm v1$) : `M-sub`($\pm v2$)) **or** $faulty_p$(mgr);
    if (!FAULTY$_p$(mgr))
        DoPreCommit($v2$);
        DoCommit($v1$, $\pm$);
        $v1 \leftarrow v2$;
    else Wait-Reconfiguration();
until ($v1$ = nil-id);
/* end Outer Process Task */

Reconfiguration

Let p have local version $x - 1$. For Reconfiguring, we use the following variables:

- **NextUpdate$_p$** is a tuple of the form $[< v, x >, i]_p$, where $< v, x >$ is the value p is waiting to commit to form LocalView$_p^x$, and rank(i) is the rank (in LocalView$_p^{x-1}$) of the initiator that submitted $< v, x >$. When p receives a submission it changes **NextUpdate$_p$** to reflect the value proposed and the initiator proposing it.

- **LastCommit$_p$** is value p committed to form LocalView$_p^{x-1}$.

- $state(p)$ is the triple [ver(p), NextUpdate$_p$, LastCommit$_p$].

- **Ahead$_r$** is the set values reported committed for versions numbered greater than ver(r). Initiator r receives these values in response to its `R-int`($state(r)$) message. In actuality, the only reported version in Ahead$_r$can be ver(r) + 1.

- **SubCurrent$_r$** is the set of **NextUpdate** values r receives with proposed versions equal to ver(r) + 1; **SubAhead$_r$** is the set with proposed versions greater than ver(r) + 1.

Task: Reconfiguration Initiator, r, with ver(r) $= x$

$Mcast_r(\mathsf{LocalView}_r, \texttt{R-int}(state(r)))$;
forall $p \in \mathsf{LocalView}_r$
  await either $recv_r(p, state(p))$ **or** $faulty_r(p)$;
if (majority of $\mathsf{LocalView}_r$ didn't respond) $crash_r$;

/* Determine the value and version to submit from the responses received. */
DetermineProposal($v1$, ver, $v2$);
DoPreCommit($v1$);
$Mcast_r(\mathsf{LocalView}_r, \texttt{R-sub}(< \pm v1, ver >))$;
forall $p \in \mathsf{LocalView}_r$
  await either $recv_r(p, \mathsf{ack}(\texttt{R-sub}(< \pm v1, ver >)))$ **or** $faulty_r(p)$;
if (majority of $\mathsf{LocalView}_r$ didn't respond) $crash_r$;

DoCommit($v1$);
if (Joining new members)
  $Mcast_r(v1, \texttt{Join} : \texttt{State-Xfer})$;
  forall $p' \in v1$
    await either $recv_r(v1, \mathsf{ack}(\texttt{Join}) : \mathsf{NextUpdate}_{p'})$ **or** $faulty_r(p')$;
  if $(\mathsf{NextUpdate}_{v1} \neq \perp)$
    $v2 \leftarrow \mathsf{NextUpdate}_{v1}$;

$Mcast_r(\mathsf{LocalView}_r, \texttt{R-com}(< \pm v1, ver >) : \texttt{R-sub}(\pm v2))$;
mgr, $v1 \leftarrow r$, $v2$;
Begin mgr Task;

Task: Outer Reconfiguration, p

```
recv_p(r, R-int(state())r);
if (rank(p) > rank(r))
    crash_p

/* Catch up to r if necessary */
if (ver(p) < ver(r))
    DoCommit(LastCommit_r);
    state(p) ← state(r);

send_p(r, state(p));
await either recv_p(r, R-sub(< ±v1, ver >)) or faulty_p(r);

if (not FAULTY_p(r))
    DoPreCommit(v1);
    send_p(r, ack(R-sub(< ±v1, ver >)));
    await either recv_p(r, R-com(< ±v1, ver >) : M-sub(±v2)) or faulty_p(r);

    if (not FAULTY_p(r))
        DoCommit(v1);
        mgr , v1 ← r, v2;
    else Wait-Reconfiguration();
else Wait-Reconfiguration();
```

```
/* Sets parameters proposal, version, and invisible. Let ver(r) = x.  */
Procedure: DetermineProposal(OUT < proposal, version >, OUT invisible);

Ahead_r ←       {[Ids, ver(p)]_p | ver(p) = (x + 1)} ;
SubAhead_r ←    {[Ids, ver(p) + 1, rank(init)]_p | ver(p) = (x + 1)} ;
SubCurrent_r ←  {[Ids, ver(p) + 1, rank(init)]_p | ver(p) = x} ;
if (Ahead_r ≠ ∅)
     /* Partially committed version x + 1. */
    proposal ← Ahead_r;
    GetStableProposal(invisible, SubAhead_r);
    version ← x + 1;
    return();

/* All respondents report the same local version. */
version ← x + 1;
if (SubCurrent_r is empty)
    proposal ← < −mgr , x + 1 >;
    GetUpdate(invisible);
    return();
if (SubCurrent_r is a singleton)
    proposal ← SubCurrent_r;
    GetUpdate(invisible);
    return();
/* SubCurrent_r has two elements. */
GetStableProposal(proposal, SubCurrent_r);
GetUpdate(invisible);
return();

/* update-set has no more than two elements. */
Procedure: GetStableProposal(OUT < val, ver >, IN update-set)

< val, ver > ← the element of update-set with the lowest ranked initiator.
return();
```