The Group Membership Problem in Asynchronous Systems

Aleta Marie Ricciardi
Ph.D Thesis

92-1313
November 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
THE GROUP MEMBERSHIP PROBLEM IN
ASYNCHRONOUS SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School
of Cornell University

in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by

Aleta Marie Ricciardi

January 1993
THE GROUP MEMBERSHIP PROBLEM IN ASYNCHRONOUS SYSTEMS

Aleta Marie Ricciardi, Ph.D.
Cornell University 1993

The thesis formally defines the class of Process Group Membership Problems (GMP) for asynchronous systems. These problems involve maintaining a list of processes belonging to the system, and updating it as processes join (are started) and leave (terminate or fail). We investigate closely the strongest member of the GMP class. Strong GMP presents this list in a consistent manner to all processes using it: the sequence of joins and leaves are identical. We show that despite prevalent beliefs, strong consistency and efficiency are not conflicting goals. This should have significant implications for distributed systems since the need for process membership agreement arises in many canonical problems in distributed computing.

We present an inexpensive means (the S-GMP algorithm) of assuring complete, system-wide agreement on process membership. We discuss the role of process membership in distributed systems and how to use S-GMP to build a Membership Resource Manager (MRM).
The thesis also examines whether any weaker member of the GMP class suffices to specify an MRM. In doing so we justify using Strong GMP over two much weaker GMP instances in three important ways. First, by comparing Strong GMP and its minimal solution with the weaker instances and their minimal solutions, we arrive at the surprising result that Strong GMP is often less expensive than the others, notably in executions in which membership changes are frequent. Second, we show that a membership service defined by Strong GMP is more robust, more responsive, and more adaptable than a membership service defined by weaker GMP instances. Third, we compare membership services defined by the various GMPs according to the utility each service provides higher-level, distributed applications. That is, ignoring implementation costs, how useful are the different GMP consistency guarantees as a platform on which to build distributed solutions to distributed problems? We show that the consistency guarantees of Strong GMP make it (i.e. the membership service Strong GMP defines) more useful to higher-level distributed applications.

Finally the thesis presents experimental results from implementing S-GMP. The data demonstrate that a centralized Membership Resource Manager is a non-intrusive service around which to design distributed systems and provide system-wide consistency. The data quantify and help clarify the tradeoffs between replication degree, overall system size, and process failure frequency. These initial results should guide future MRM design and development.
Biographical Sketch

Aleta Ricciardi was born in the wilds of the Bronx, NY – without doubt responsible for her love of outdoor sports. Her future as a computer scientist was fated when her father brought home a vintage IBM teletype circa 1967 and made a valiant stab at teaching her Fortran, Cobol, and all about π. She raised more than her fair share of aitch-ee-double-ell in her formative years (which, according to her parents, lasted far longer than the national average) but nonetheless managed to graduate from Cornell University in 1984, taking refuge in Biology and Mathematics until key punches had been relegated to the dustbin of history. Meeting Prakash Panangaden on a porch somewhere, some night, in Collegetown awoke her early memories of Computer Science and she left her wild days for the monastic existence of graduate school. As is only fitting, she met Michael while cycling around Cayuga Lake on the only nice day of the Summer of ‘92.
To my parents, Marie and Emil, and to Bugs Bunny.
Acknowledgements

This work would not have been possible without the guidance, support, enthusiasm, and generosity of Ken Birman. Ken’s confidence has been a constant source of inspiration, and his patience in guiding my evolution has been remarkable. Keith Marzullo, too, has been both friend and advisor — I have benefited greatly from (and enjoyed infinitely) our many discussions, technical and political, from the lunatic fringe. Without Ken’s and Keith’s insights, warmth, and genuine concern my experiences here would have been severely diminished. Prakash Panangaden and Bob Constable gave me early encouragement and guidance, as well as the initial desire to pursue these studies.

Robbert van Renesse has been invaluable in all technical matters. More than any other, Robbert’s assistance in the implementation stage was unflinching and enthusiastic; I have learned a great deal from him. I have also benefited from many discussions (and bagels) with Pat Stephenson, Brad Glade, Mike Reiter, and the rest of the Isis crowd who have been both friends and helpful colleagues. Tom Bressoud has been the perfect office-mate; I cannot begin to thank him for his patience, knowledge, and for laughing at all my dumb jokes. David Cooper proof-
read the entire thesis and gave many valuable comment. Many thanks also to our technical support staff, especially John Finley, Doug Flanagan, Anne Louise Gockel, and O.J.

I have also been fortunate to have received funding from IBM, the Siemens Corporation, NASA, and General Electric.

Bruce Donald has been a continual joy – a dear friend whose kindness and sense of humor have added brilliantly to my life. From Dan Huttenlocher I have gotten warmth, laughter, generosity, and a great deal of confidence. Daniela Rus, in addition to her more obvious world-class talents, is also a wonderful listener and ‘best woman’. Michelle Duncan and I have been through many interesting and important events – quite fittingly, all electronically. Thanks, too, to ‘The Women’s Cabal’.

Joseph Moore was a tremendous source of comfort; he made sure my life was as filled with love, fun, and music as it was with work – no small task. Mari Orser and Avigail Eisenberg, too, have been dear and cherished friends. Susie “Fortebraccie” Armstrong and I have logged many kilometers in the saddle and enjoyed every moment, which kind of goes without saying.

In the “Moldy Oldies” category are Deirdre Taub and Nathan Yoffa – both have helped and watched me grow, and made me laugh for many, many years. Chuck Fuller and Mark Theodore also occupy very special places.

Above all my family, especially my parents and brothers, have given me more support, encouragement, and inspiration than I believe they’ll ever understand. I have felt equally and constantly their faith in me, their love, and their pride in my
accomplishments.

Finally, Michael Ogg has made what should have been a very trying period, so very beautiful and exciting. He has shown me such wondrous joys I often find myself beyond words.
# Table of Contents

1 Introduction .......................... 1
   1.1 Examples ........................................ 3
   1.2 Membership Consistency and Distributed System Design .......... 7
      1.2.1 Implications for Distributed Systems ................. 8
      1.2.2 MRM Design Considerations .................. 8
   1.3 Related Work ......................... 11
   1.4 Thesis Contributions ................... 16
   1.5 Thesis Organization ................... 19

2 The System Model and Formal Logic .......... 20
   2.1 System Requirements and Model Assumptions .............. 21
   2.2 The System Model ................................ 22
   2.3 The Modal Logic $\mathcal{L}_{async}$ .............. 26
      2.3.1 Syntax of $\mathcal{L}_{async}$ .................. 28
      2.3.2 Semantics of $\mathcal{L}_{async}$ ............... 29
      2.3.3 Inference Rules and Axioms ................ 30
3 Strong Group Membership 32

3.1 Formal Specification ........................................... 33
3.1.1 More About $GpView_{\Sigma}(e)$ .......................... 35
3.1.2 Strong GMP Specification .................................... 36

4 A Protocol Solving Strong GMP 40

4.1 Simple S-GMP .................................................... 41
4.2 Full S-GMP ....................................................... 44
4.2.1 Structure of the Reconfiguration Algorithm .............. 46
4.2.2 Rules of Succession .......................................... 48
4.2.3 Rules of Progression: Maintaining Safety ............... 49
4.2.4 Reconfiguration Phase I Responses ......................... 51
4.3 The S-GMP Algorithm .......................................... 53

5 Correctness 65

5.1 The Inductive Step .............................................. 65
5.2 S-GMP Correctness Proofs .................................... 87
5.3 Message Complexity ............................................. 94

6 Protocol Optimality 97

6.1 Reformulating Strong GMP ..................................... 99
6.2 Knowledge Analyses ............................................ 102
6.2.1 Facts from Correctness .................................... 105
6.2.2 Knowing Stability ........................................... 106
6.2.3 Optimizations to S-GMP ..................................... 110
6.3 Optimized s-GMP ........................................ 115
6.4 Message-Minimality ....................................... 119
6.5 Versions Without Clean Starting Points .............. 121
6.6 Epistemic Quantification of Strong GMP .............. 125

7 The Role of Order ........................................ 129

7.1 Characteristic Membership Properties ................. 129

7.2 Three MRM Services .................................... 132
  7.2.1 Virtual Partitions ................................... 133
  7.2.2 The VP Algorithm ................................... 133
  7.2.3 Eventual Propagation Only ......................... 136
  7.2.4 The EP Algorithm ................................... 137
  7.2.5 Strong GMP .......................................... 139

7.3 Message Complexity Comparison ....................... 140

7.4 Using a Membership Service ............................. 142
  7.4.1 Virtual Partitions ................................... 143
  7.4.2 Eventual Propagation ................................ 144
  7.4.3 Strong GMP .......................................... 145

8 Implementation and Performance ......................... 148

8.1 Measuring Performance ................................... 149

8.2 Physical System ......................................... 151
  8.2.1 Network Layer and Machines ....................... 151
  8.2.2 The Name Service and MRM Start-up ................ 152
8.3 The Client-Server Interface ........................................ 153
  8.3.1 Requests and Event Notifications ............................... 155
8.4 Detecting Failures in the MRM ...................................... 157
8.5 Testing the MRM ....................................................... 158
  8.5.1 Parameters and Expectations .................................... 158
  8.5.2 Testing Performance and Scalability ......................... 160
8.6 Analysis ..................................................................... 168
  8.6.1 open() versus remove() ........................................... 168
  8.6.2 Increasing Relevant-Event Frequency ......................... 168
  8.6.3 Clients (Processes) Per Site .................................... 169
  8.6.4 Groups (Threads) Per Client .................................... 170
  8.6.5 Internal versus External Response Time .................... 170
  8.6.6 MRM Size .......................................................... 171
8.7 Future Work ........................................................... 171

9 Conclusions ................................................................ 175
  9.1 Summary and Discussion ............................................ 175
  9.2 Future Work ........................................................... 176

Bibliography ................................................................ 178
List of Tables

4.1 Initiating Reconfiguration: $\text{FAULTY}_p(mgr) \land \text{FAULTY}_q(mgr)$. . . . 49

7.1 Minimal Message Cost for a Single Update to a Local View. . . . . 141
7.2 Internal Efficiency: Maximum Size Change to Local Views. . . . . 141
7.3 Utility to Applications of MRM Guarantees . . . . . . . . . . . . . . 144

8.1 Median External Response Times(ms) - MRM of size 1 . . . . . . . . 162
8.2 Median Internal Response Times(ms) - MRM of size 1 . . . . . . . . 163
8.3 Median External Response Times(ms) - $c = 5$ . . . . . . . . . . . . 172
List of Figures

1.1 Different Local Views of System’s Processes Give Incorrect Primary-Backup Scheme: there is no $a$-primary, and two $b$-primaries. .... 4

2.1 Causally Consistent and Inconsistent Cuts. ................. 25

2.2 Causal Binary Relations on Consistent Cuts. ................. 27

4.1 Two-Phase Communication Structure of Simple S-GMP. ........ 42

4.2 Compressing Successive Instances of Simple S-GMP. ........ 43

4.3 $mgr$’s Failure Results in Undefined Group View. ............ 45

4.4 Reconfigurer $r_2$ Learns of Conflicting Proposals for $GpView^x$. 47

4.5 Majority of Responses Needed ............................ 50

4.6 Removal of $q$ Committed Invisibly to $p$ and $r$ .............. 51

4.7 A Situation Requiring New Core Members to Report $NextUpdate$ to Initiator. ........................................ 55

5.1 Case Analysis for Proposition 5.1.7 , Part 1. ................. 76

5.2 Case Analysis for Proposition 5.1.7 , Part 2. ................. 77

5.3 Continued Case Analysis for Proposition 5.1.7 , Part 2. ....... 78
5.4 Proving $c_x$ Consistent – Theorem 5.2.4  

5.5 Proving $c_x$ Consistent – Theorem 5.2.4 continued.  

6.1 Double Conmit Optimizations  

7.1 The Virtual Paritions Algorithm.  

7.2 The Eventual Propagation Algorithm.  

8.1 External and Internal Response Time  

8.2 Effect of Increasing Relevant-Event Frequency on External Response Time.  

8.3 Effect of Increasing Relevant-Event Frequency on Internal Response Time.  

8.4 Effect of Heavy-Weight Context Switch on External Response Time.  

8.5 Effect of Light-Weight Context Switch on External Response Time.  

xiv
Chapter 1

Introduction

"The election protocol is straightforward. All sites involved in the election (i.e. all those which are operational) agree on a linear ordering of sites and the first site in this ordering is elected as the new coordinator."

David Bell and Jane Grimson in Distributed Database Systems

The following 177 pages are straightforward.

This thesis investigates the problem of process membership in asynchronous distributed systems. This problem consists of maintaining a list of processes belonging to the system, and updating it as processes join (are started) and leave (terminate or fail). A key goal is that the list be presented in a consistent manner to all processes using it: the sequence of joins and leaves should be identical. Consistency is a strong property and might be expected to come at a high price but, somewhat surprisingly, this is not the case.
A distributed system consists of a set of independent and geographically distinct processors together with some means by which they communicate. Distributed systems offer substantial benefits, for example increased availability and performance, over non-distributed (or isolated) systems. One obtains these benefits by exploiting replication, locality, and concurrency. Unfortunately, writing programs that take full advantage of these properties is quite difficult since there will be many independently-executing processes whose local states and possible interactions must be understood. Moreover, in many distributed systems there are no timing guarantees so that both combinatoric complexity and system asynchrony combine to prevent users from realizing the gains originally promised by distributed environments. However with the appropriate formal tools we can better understand these complex interactions and how they affect the problem at hand.

Experience in the field shows there are a number of recurring themes, similar goals, and common sources of difficulty in solving problems. These provide reliable hints for constructing the desired formal tools. Process groups, in particular, have proven to be an especially useful and natural paradigm for programming in and reasoning about asynchronous distributed systems [CZ85,Coo85,BJ87,PBS89]. Process groups arise, among other cases, whenever a process is replicated to provide fault-tolerance, when a set of processes cooperate to execute a distributed event, share memory, subdivide a computation, and so forth. These problems are seen in database contexts [BH87a], real-time settings [CT90], and distributed control applications [MBCW90]. The particular use of a process group determines its required behavior and semantics: in some settings group members may always need to know
the exact composition of the group, while other settings do not require such strict coordination. The class of Process Group Membership Problems describes the range of desired group semantics according to, among other things, the members' level of agreement regarding the group's composition and the amount of coordination required to change the membership. Given a process group membership problem, a group membership service monitors process groups for clients, ensuring they observe only the desired group semantics [Cri90, RB91, CT91, MS92b]. As a simple example, a linear group membership service would never report that two members left a group concurrently; it would report one's departure before the other's, imposing an artificial order whenever necessary. Similarly, a complete group membership service would ensure that all clients interested in a particular group see the same set of changes to that group, though not necessarily in the same order.

Most group semantics require processes to be removed from the group when they fail and permit them to rejoin when they recover. Thus, the membership service must have some means of detecting and reporting processes' failures and recoveries. Moreover because it is a system service, it should be designed fault-tolerantly, and this argues for replicating the membership service protocol at distinct processors. As a result a membership service is itself a process group, whose members must be informed of changes to its own composition.

1.1 Examples

---

1 Though we will use the term “recovered”, we actually model a recovered process as a completely new instance of the specified task. This simplifies our algorithms, allowing us to ignore the case of a process that fails and recovers intermittently.
Figure 1.1: Different Local Views of System’s Processes Give Incorrect Primary-Backup Scheme: there is no \(a\)-primary, and two \(b\)-primaries.
It is not hard to envision situations in which inconsistent information regarding processes' functionality might cause a replicated program to violate its specification. Consider the case of primary-backup replication methods for fault-tolerance [AD76, Bar81, BMST92]. In primary-backup, a system service has a primary server that executes independently and periodically transfers state to a designated backup server. The backup takes over for the primary when the latter fails, becoming the new primary server with its own designated backup server. Consistency requires there be at most one primary server at any given time, while availability requires at least one. In the example depicted in Figure 1.1, the initial primaries for the services \( a \) and \( b \) are, respectively, \( a_2 \) and \( b_2 \). The operating system on Machine 2 knows \( a_2 \) is failed but the operating system on Machine 1 has not detected this yet. Whereas \( a_1 \) should be acting as the \( a \)-primary it is not, leaving no \( a \)-primary. Further the operating system on Machine 1 erroneously timed out on \( b_2 \) causing \( b_1 \), the backup, to take over with the result that two \( b \)-primaries exist.

This situation can be distressingly common – whenever operating systems make decisions independently regarding the state of processes executing remotely, their individual views of the set of system processes need not agree. However, recalling the terminology above, a complete, linear membership service (i.e. one that ensures all clients interested in the same group see changes to it in the same order) would easily guarantee correct behavior in the primary-backup situation.

The Mach operating system provides another example [Ras86, ABG+86]. In Mach each port has at most one receiving task, which grants other tasks send-rights (i.e. capabilities) to that port. At each site, the \texttt{NetMsgServer} keeps track of the
ports to which its local tasks hold send rights. When the NetMsgServer detects that all local tasks holding send rights to a specific port have failed, the NetMsgServer issues a **no-more-senders** message to (the NetMsgServer of) the receiving task. Similarly, all tasks holding send-rights to a port will get a **dead-receiver** message (from the port’s local NetMsgServer) when the port’s receiving task fails.

The results of Fischer, et.al. [FLP85] imply that to support reliable, inter-process communication\(^2\) in asynchronous systems one must either sacrifice liveness to ensure safety, or sacrifice the accuracy of failure determinations to ensure liveness. MachIPC chooses the former by making failure determinations only upon the failed task’s recovery. Since all detections are made locally, they will always be accurate. However, since the various NetMsgServers do not coordinate their failure notifications, each NetMsgServer has a different view of the set of functioning and failed tasks in the entire system. Again, applications needing more stringent system membership information must build it themselves.

If programmers are unaware of the importance of the membership problem then applications may behave incorrectly and in unexpected ways. If programmers are aware of membership issues then the problem’s complexity can be daunting and prevent designers from realizing and exploiting the full computational power of distributed systems by not developing fully-distributed solutions. To this end, a system-wide service providing consistent information on processes’ functionality would greatly facilitate writing reliable, distributed programs.

\(^2\)"In the absence of failures, all messages are eventually delivered". Unfortunately this is an historically vague description, and interpretation varies from one implementation to another."
1.2 Membership Consistency and Distributed System Design

We now briefly discuss the implications of guaranteed system membership agreement for distributed systems’ design. Earlier in this chapter we described the aims of and need for a process group membership service. To emphasize that process management is a basic system concern we propose and discuss a new system service, the Membership Resource Manager (MRM). While process group membership information is important to many distributed applications, a general MRM service stresses the fundamental role of task management in any system (distributed or otherwise), as well as our more specific belief that it is intrinsically necessary for distributed systems’ consistency.

System membership consistency is a conspicuous difference between distributed and non-distributed operating systems. Non-distributed operating systems provide complete consistency with respect to process management – in fact an operating system defines the set of executing processes. By viewing processes as system resources, it is only natural that managing them is an operating system’s responsibility. In contrast, distributed systems consist of a loosely-coupled collection of autonomous, and often different, operating systems. While each autonomous O/S manages and defines its local process set consistently, it can only approximate the set of remote processes. As a result the autonomous systems’ local views about global system composition may conflict at any point in time. A membership service, upon which all system components rely for membership information, will preclude this conflict.
Moreover by mimicking the membership consistency of non-distributed systems an MRM will greatly simplify programming in distributed environments.

1.2.1 Implications for Distributed Systems

System membership consistency imposes a number of restrictions on processes. One is that all system processes must be known to the MRM; a process or its host operating system must inform the MRM of its birth. Another restriction is that all processes making decision based on system composition must become clients of the MRM – even if the decision is based solely on local processes. While the MRM can attempt to respect the actual order of births and failures, it may not be able to do so in all cases and so a process that relies on its host O/S for membership information may receive information that is not globally consistent. We examine these restrictions and others more closely in Chapter 8.

1.2.2 MRM Design Considerations

The following is a brief overview of the major issues in designing an MRM. We discuss them in greater detail later, but it will be beneficial to keep them in mind during the theoretical discussions.

1.2.2.1 Fault-tolerance

Because the MRM will be a central system service, it should not have a single point of failure. Achieving permanent availability is impossible – we cannot prevent catastrophic failures – but we can achieve a high degree of fault-tolerance by replicating the protocol and executing the MRM on different processors. However,
we cannot do this naively: for the MRM to provide consistency, it must provide the same behavior despite replication, and despite the failure of processors on which the MRM protocol replicas are executing. The MRM, although implemented by a process group, must appear to be a single, fault-tolerant process. *Strong GMP* is the particular Process Group Membership problem that corresponds to these desired group semantics; we discuss this in depth in Chapter 3.

We call the processes executing the MRM protocol the *core members*. The assumption is that this set can maintain a replicated database listing other processes in the system. Our focus in this dissertation is on the protocol run by the core members. Both Strong GMP and underlying system properties determine the number of simultaneous core member failures the MRM can tolerate for a given core size. In the final analysis, this and the experimentally observed performance considerations determine the actual degree of replication needed.

### 1.2.2.2 Performance and Scale

Since all system components will be clients of the MRM, performance is crucial. While it is commonly assumed that fault-tolerance and performance are conflicting goals, an important result of our work is that this is not the case. It turns out that Strong GMP's strict consistency guarantees are symbiotic: the effect of their combined presence is to allow the protocol solving Strong GMP to accomplish more with less effort than protocols solving weaker group membership problems.

How well the MRM scales will be affected, primarily, by MRM replication, overall system size, and the frequency with which system processes fail and recover.
Increasing the latter two stresses the MRM correspondingly. While the MRM can add new core members to relieve the stress, the larger core will also be slower.

In a large system we would design the MRM hierarchically, drawing the separation lines according to the following schemes:

- Notice that having implemented the single, fault-tolerant process illusion, we can obtain the same strong ordering guarantees over a very large set of processes using a simple reliable multicast. In this way, the actual protocol solving Strong GMP would only be run by a small set of processes which would maintain the fault-tolerant membership list for the overall system.

- As will become clear in the following discussion the MRM’s true purpose is ordering process failures and recoveries. However, implementing the full service requires a number of tasks that are unrelated to and independent of ordering. These can be done by distinct services, greatly reducing the core members’ load.

1.2.2.3 Client-Server Interface

A client needs some means of registering with the MRM and of specifying the entities (e.g. process groups, machines) it wants monitored. Client registration is among those functions that are not essential to ordering and it need not be administered by any core member. In large systems, scalability concerns will argue strongly for dedicating MRM core members specifically to ordering, so we would implement a distinct registration service.
Each client must also be informed of the failures of entities the MRM is monitoring for it. How the MRM reports this information is a difficult issue and may be best handled by yet another distinct service. In small systems, a simple round-robin or hashing scheme would distribute reporting responsibility equally among the core members. To ensure updates are delivered in the correct order each must contain a unique, incremental sequence number. In a large system, having the MRM dispatch its updates to a dedicated reporting service, rather than to the entire system, would improve the core's scalability.

1.2.2.4 Information Gathering

Finally the MRM must gather relevant information on the entities it monitors. Among other means, it can get hints from the autonomous operating systems within the distributed system, and it can ‘ping’ entities periodically. However, one could argue that the essential function of the MRM is to order failures and recoveries, not detect them. Thus, in large systems still another, distinct service will improve performance and scale significantly. A detecting service could be quite sophisticated, for example by filtering out duplicate hints, or by ignoring provably erroneous hints.

1.3 Related Work

A number of other efforts have addressed the problem of process group membership, though all differ from the work presented in this thesis in the assumed model of computation (synchronous versus asynchronous), consistency properties desired of group membership (particularly the observed order of changes), and/or protocol
structure (ours is multi-phase and asymmetric – a distinguished process coordinates changes).

The work most similar to this is the Isis system’s site view management protocol [BJ87]. The fundamental abstraction in Isis is virtual synchrony – the inability to distinguish an execution in a synchronous environment from one in the Isis environment. Isis constructs virtual synchrony hierarchically, using system membership information to build multicast primitives that are atomic (“all members of a destination group deliver the multicast message or none do”) and ordered (both causally and totally). Atomicity is vital to the virtual synchrony illusion, but is particularly tricky when the destination group changes. Building the multicast primitives on a strong membership service simplifies matters.

Like our membership protocol, the site view management protocol is asymmetric and multi-phase. It is invoked whenever the view manager (i.e. coordinator) detects a view member’s failure or the recovery of a former member. The current Isis site view update protocol is not live when view members fail and recover frequently; the protocol completes only if there are no additional failures and recoveries while it is executing. In the same situation our protocol achieves its best performance, making rapid and efficient progress.

The current Isis site view management protocol uses mandatory quorum consent to ensure the same strong consistency guarantees as ours – every site sees the same sequence of site views in the same order. Implementing the various multicast protocols on top of the site view management makes them simpler. Since changes to the destination group are well-defined, the protocols can use the composition of
the destination group in determining the correct delivery order.

Also in the asynchronous domain Misra, et al. [MS92b] defined a membership service built on top of the Psync multicast mechanism [PBS89]. This mechanism maintains a partial order, based on causality, on message delivery, and does so despite processor failures. The membership consistency guarantees are weaker than ours, ensuring only that all processes observe all causally-related failures in the same order; it makes no attempt to order concurrent failures. As a result, their protocol admits more concurrency than ours but sacrifices the benefits of our stronger consistency guarantees. The structure of their protocol also differs markedly from ours – using the Psync multicast and its guarantees, processes make decisions to change their local membership views independently. Independence, however, is expensive, in total messages as well as the delay encountered before a membership change becomes stable (i.e. known and correctly ordered by all processes).

Moser, et al [MSMA91] also took this approach, building a membership service on top of their Trans and Total partially- and totally-ordered multicasts [MSMA90]. In both these instances one should note that, because the notion of a process group is not defined at the multicast layer, every system process must be included in every multicast. This contrasts to our use of a core set of processes that run the protocol.

The Chang and Maxemchuk [CM84] atomic broadcast protocol contains an implicit solution to the membership problem.\(^3\) Their broadcast protocol uses knowledge of the destination group’s composition to ensure message ordering, making agreement on the status of these processes important. This is interesting because we

\(^3\) Marzullo [Mar] has identified an error in their reformation protocol, arising from incompletely specifying the number of processes that must consent to the new token holder’s identity.
now see three approaches to the same (or at least quite similar in the case of Psync) distributed problem that make clear how deeply embedded group membership can be. Psync and Total (lacking a formal notion of process group) must include every system process as destinations, the Chang-Maxemchuk protocol solves membership while solving atomic broadcast, and Isis solves membership first, implementing the ordered multicast on top.

Still in the asynchronous domain, the Virtual Partitions protocol of [ASC85] (VP) provides very weak consistency guarantees, in fact the weakest we formulate. The sole guarantee made by VP is that if a system ever becomes quiescent (failures and recoveries neither occur nor are suspected), eventually processes’ local views will be identical. In the absence of quiescence, local views need never agree. This guarantee, however, seems more a statement about how processes detect failures. Specifically, it requires that all crashes be detected, and also that erroneous suspicions cease. Like our protocol, the VP protocol is multi-phase and asymmetric, but since VP makes neither ordering nor uniqueness guarantees an update coordinator is not bound to obtain any particular degree of consent. As we discuss in Chapter 7, this turns out to have significant ramifications for VP’s utility as a membership service.

More abstractly, the Distributed Consensus (DC) impossibility result [FLP85], would appear to preclude solving Strong GMP. Just as Strong GMP requires processes to agree on each change to the group, DC requires all non-faulty processes to decide on a single value. Fischer, Lynch, and Paterson proved this cannot be achieved in asynchronous systems that experience even one crash failure. However,
while the goals of Strong GMP and DC are similar, Strong GMP differs in that it will exempt some non-faulty processes from deciding; that is, Strong GMP allows non-faulty processes to be classified incorrectly as faulty. In fact, the impossibility of distinguishing crashed from slow processes is precisely the reason DC is unsolvable in asynchronous systems. Chandra and Toueg proved that failure detectors characterized by surprisingly weak properties suffice to solve Distributed Consensus [CT91]. One of these properties is Weak Accuracy which requires that there be some point in an execution after which some correct process is no longer erroneously suspected of having failed. Weak Accuracy formalizes the source of DC’s impossibility – an asynchronous system can never guarantee this behavior.

In the synchronous model of computation failure detections are accurate and can be agreed upon independently and, if not simultaneously, then within a small, bounded time interval. In contrast then with asynchronous systems, there is no need to worry about actions taken by processes that have been deemed faulty incorrectly. Cristian’s[Cri90] membership problem and solutions focus, accordingly, on timeliness guarantees on membership changes. These are determined by the system’s a priori bounded clock drift and message delivery parameters. Nonetheless, timing failures are possible and are treated as true crash failures. In particular, to ensure the timeliness guarantees, late messages are ignored and the sender, once having violated a process correctness criterion, is deemed faulty.
1.4 Thesis Contributions

This thesis shows that, despite prevalent beliefs, consistency and efficiency in distributed systems are not conflicting goals. While it is often assumed that inter-process consistency and coordination cannot be provided cheaply, we present an inexpensive means of assuring complete, system-wide agreement on process membership. This can have significant implications for distributed systems since the need for process membership agreement arises in many canonical problems in distributed computing. Being able to provide membership consistency cheaply will greatly simplify solving these problems.

The thesis formally defines the class of Process Group Membership Problems for asynchronous systems. This formalization delineates a hierarchy of membership problems, and the resulting distinction simplifies comparing previous approaches to Process Group Membership. We focus on the strongest of the group membership problems, Strong GMP, the only version of the class that requires all participants to observe changes in a group’s composition in the same order. The thesis develops the S-GMP protocol\(^4\) and proves it correctly solves Strong GMP. We also prove S-GMP minimal in the number of messages and phases of communication it uses; that is, no other protocol can solve Strong GMP in fewer messages or fewer phases than S-GMP does.

We then examine whether any of the weaker versions of GMP suffice to specify a Membership Resource Manager, and in doing so, justify using Strong GMP over two less stringent GMP instances in three important ways. First, by comparing Strong

\(^4\)For Strong Group Membership Protocol
GMP and its minimal solution with weaker GMP instances and their minimal solutions, we arrive at the surprising result that Strong GMP is often less expensive than the others, notably in executions in which membership changes are frequent. In this way we refute the common intuition that strong consistency guarantees are unreasonably expensive.

Second, we show that a membership service defined by Strong GMP is more robust, more responsive, and more adaptable than a membership service defined by weaker GMP instances. Specifically, at any point in time the Strong GMP service can tolerate almost half its current members failing simultaneously; a weaker service can tolerate only a fixed number of simultaneous failures (usually no more than two). Moreover, in contrast to the weaker membership services, the Strong GMP service can change its own composition radically to adapt to sudden changes in its work load – it can instantly add an arbitrary number of new components to help share a dramatically-increased load, or instantly remove nearly half its component servers if its load drops. Again, a weaker service can only add or delete one or two at a time.

Third, we compare the various GMPs according to the utility each provides higher-level, distributed applications. That is, ignoring implementation costs, how useful are the consistency guarantees of the various GMPs as a platform on which to build solutions to distributed problems? We define utility in relative terms: we measure the complexity of an application when it is built on top of each of the three membership services. Then by comparing the overall costs, we approximate the amount of extra work an application incurs when it is built on top of one.
service versus another. Within this framework we prove a Strong GMP membership service is a much more useful foundation for building distributed applications. To do this, we rephrase the original formulation for the class of Group Membership Problems to emphasize its role as a failure model. Given an asynchronous system in which processes experience crash failures,\(^5\) Strong GMP simulates a Weak Fail-Stop failure model [SS83, Sch84, MS92a] in which all system processes also agree on the order of failures. Weak Fail-Stop is a very benign failure mode, making systems that experience this failure behavior highly desirable. The failure mode semantics simulated by the weaker GMP instances are not nearly as useful. In fact the two weaker versions we examine provide virtually no relevant information to the application we considered (managing a single-copy replicated data item)

This thesis also contains experimental results from implementing S-GMP. By implementing S-GMP and testing it extensively we demonstrate that a centralized Membership Resource Manager is a useful, non-intrusive foundation upon which to build system-wide consistency, thereby backing up the theoretical claims made earlier. Moreover the data quantify and help clarify the tradeoffs between replication degree, overall system size, and process failure frequency. These initial results will be useful for designing and developing future MRMs.

The method we used to prove S-GMP minimal applies techniques developed by Chandy and Misra [CM85] and furthered by Mazer [Maz89]. By formulating Strong GMP in terms of process knowledge acquisition, we found optimizations to the original S-GMP protocol, and proved the new protocol message-minimal. In

\(^5\)We suspect the result holds for omission failures as well.
this way, the thesis expands the practical uses of epistemic logics in solving and reasoning about inherently distributed problems.

Finally, this dissertation attests to the importance of specifying a problem accurately, as well as the inherent relationship between problem specification and problem solving. The results outlined show the same problem can and should be specified in different ways depending upon which aspect of the problem one wishes to examine.

1.5 Thesis Organization

Chapter 2 formally describes our computational model – an asynchronous distributed system in which component processes may experience crash failures – and the epistemic and temporal logic derived from it. We use this logic in Chapter 3 to define the Strong Group Membership Problem. Chapter 4 presents a solution, the S-GMP algorithm, to Strong GMP which we prove correct in Chapter 5. In Chapter 6 we derive optimizations for S-GMP and prove the resulting algorithm optimal with respect to message complexity. Chapter 7 justifies using Strong GMP to specify a MRM by comparing Strong GMP with weaker versions. Chapter 8 discusses our implementation of Strong GMP and presents performance results. We conclude in Chapter 9.
Chapter 2

The System Model and Formal Logic

The dissertation considers only asynchronous distributed systems in which processes fail by crashing. Specifically, *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, especially given the overriding global reliability and consistency concerns of this thesis. System load, network traffic, and any other dynamic components of the system that affect performance all conspire to violate synchronization assumptions. By assuming asynchrony, we solve the broader problem and are assured of correctness in all system executions, not just those in
which certain timing constraints are met. In fact, in systems with synchronous assumptions, the possibility of experiencing a *timing failure* is often made explicit.

### 2.1 System Requirements and Model Assumptions

We want to clarify the distinction between system requirements and model assumptions. The former are concrete, fundamental properties of the system upon which our results depend. The latter are abstract, higher-level properties or utilities that can be implemented given the system requirements. We adopt model assumptions because they simplify reasoning about the class of Group Membership Problems by masking system details that we believe obscure the aspects of GMP we wish to explore.

We require that each message sent along a channel has a non-zero probability of reaching its destination intact. With this basic requirement one can build a point-to-point, completely-connected communication network that eventually (and exactly once) delivers uncorrupted messages in FIFO order (*e.g.*, van Renesses's Multicast Transport System [vRBC+92]). We assume this more sophisticated package for our model's communication network; we are not concerned with how the complete communication package is implemented, but our protocols will assume these guarantees.

We also require that each process has a local, monotonically increasing clock (or event counter) and that processes fail only by crashing. System asynchrony makes
crash failures impossible to detect accurately: a delayed response, a slow process, and a crashed process are indistinguishable. Nonetheless, our model assumes that there is some means by which one process comes to suspect another faulty. We permit incorrect failure suspicions, but assume that each process’s failure suspector eventually suspects a crashed process. Local counters coupled with our model’s communication system suffice to implement this assumption. Processes do not reverse their failure suspicions.

We further assume that a process disconnects itself from those it suspects faulty and so receives no further messages from them (for example, a process may close its incoming channel). A process also gossips (for example, with piggy-backs) its failure suspicions to other processes in all communication, whereupon the recipient adopts the sender’s belief.\footnote{There is no harm in a process believing itself faulty through gossip.} The gossip and disconnect assumptions tend to isolate suspected-faulty processes among those with mutual non-failure beliefs; that is, among all processes not believing each other faulty. Notice that one process’s beliefs affect another’s behavior only if the first sends a message to the second, and only if the second does not already believe the first faulty.

\section{The System Model}

Denote by \textbf{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 ever 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. A process’s state consists of the values of all its local variables, together with the set of messages it has sent and received; an event executed by a process results in a state transition by that process. 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 quit\(_p\) models the crash failure of process p, after which only other quit\(_p\) events are permitted.\(^2\) 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\(_p\):

\[
h_p \overset{\text{def}}{=} \left< \text{start}_p \cdot e_p^1 \cdot e_p^2 \cdot \ldots \cdot e_p^k \right> \quad k > 0.
\]

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 \(p_i \in \text{Proc}\). We restrict our attention to cuts determined by finite subsets of \(\text{Proc}\) since these represent the system’s global system state at some real time in its execution. Each execution begins with the distinct cut, \(c_0 = < \text{start}_{p_1}, \text{start}_{p_2}, \ldots, \text{start}_{p_n}>\). Lacking better notation 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.

From Lamport [Lam78], we give the following definitions of event causality and global, causal consistency.

**Definition** Events e and \(e'\) are immediately causally related in cut c (written \(e \rightarrow_c e'\)) if and only if

---

\(^2\)This is purely for ease of exposition.
1. $e = e_p^i$ and $e' = e_p^{i+1}$ in some component $h_p$ of $c$; or

2. $e = send_p(q, m)$, $e' = recv_q(p, m)$, $e \in c$, and $e' \in c$.

We write $e \rightarrow_c e'$ to denote the binary relation happens before, which is the reflexive, transitive closure of $\rightarrow_c$ over the events of $c$. ■

Observe that, in the abstract, send and receive events are always (i.e., in all cuts) $\rightarrow_c$-related while other events derive the relation from their historical context. We will omit the subscript when the particular cut is clear from context and simply write $e \rightarrow e'$.

**Definition** A cut $c$ is causally consistent if and only if whenever $e' \in c$ and $e \rightarrow e'$ in $c$, then $e \in c$. ■

Figure 2.2 illustrates both a causally consistent cut, and one that does not respect causality. Henceforth we restrict the discussion to consistent cuts, as they are the ones that are physically realizable. Consistent cuts represent the possible global states of an execution; while a given consistent cut need not have existed at any point in real time in an execution, it is impossible for a cut that is not causally consistent to ever exist at any point in real time.

A characterization of global, system-wide 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 $h_p'$. 
A. Consistent Cut:
\[ \forall e \rightarrow e' : (e' \in c \Rightarrow e \in c). \]

B. Inconsistent Cut:
\[ e \rightarrow e'; e' \in c; e \notin c. \]

Figure 2.1: Causally Consistent and Inconsistent Cuts.
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, \( p \),

1. \( h_p = h'_p \); or

2. \( h_p \) is a strict prefix of \( h'_p \).

Observe that there are (infinitely) many completions (or successors) for any given cut. In this sense, the future of any cut is uncertain; it may branch out in many possible 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.

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

1. \( c \) strictly causally precedes \( c' \) (written \( c < c' \)) if and only if \( c \leq c' \) and \( c \neq c' \) (as in Figure 2.2.A);

2. \( c \) 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 2.2.B).

\[
\]

2.3 The Modal Logic \( \mathcal{L}_{async} \)

We now introduce a formal language, \( \mathcal{L}_{async} \), with which we will define the class of Asynchronous Process Group Membership problems. \( \mathcal{L}_{async} \) is a combination of temporal and epistemic logics. Unique to this combination is its attention to asynchrony - the basic semantic entities of the logic are consistent cuts. In her
A. $c \leq c'$ and $c < c'$

B. $c \ll c'$: $h_p$ is a strict prefix of $h'_p$.

Figure 2.2: Causal Binary Relations on Consistent Cuts.
doctoral thesis [Tay90], Taylor defined Concurrent Epistemic Logic. Not only are
formulae of her logic evaluated along consistent cuts, but she defines a new, weaker
modal operator that embodies system asynchrony and captures the resulting process
uncertainty.

The temporal logic component of $L_{async}$ arises from algebraically analyzing the
set of consistent cuts in an asynchronous execution, and then matching each of the
individual algebraic properties with specific characteristic axioms of tense logic.
Consistent cuts and the binary relation $\leq$ defined in Chapter 2 form a downward-
closed lattice. See [Ric90] for a proof that the tense logic arising from the corre-
spending characteristic axioms is complete for the class of Kripke models that are
downward-closed lattices.

2.3.1 Syntax of $L_{async}$

Well-formed formulae of $L_{async}$ are constructed from a countable set of primitive
propositions $\{a_1, a_2, \ldots\}$, four modal operators, and the set of process identifiers
$\text{Proc}$ as follows:

$$
\psi ::= a_i \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \Box \phi \mid \lozenge \phi \mid P_p \phi \mid K_p \phi.
$$

We use the standard abbreviations $\lor, \Leftarrow, \leftrightarrow, \Diamond$, and $\lozenge$ for readability. The English
language interpretations of the tense modalities $\Box, \lozenge, \Diamond, \lozenge$ are, respectively,
"at all future points" (henceforth), "at all points in the past", "at some future
point", and "at some point in the past". The English interpretations of the

\[^3\text{We use the duality interpretation for } \Diamond, \text{ rather than the branching time interpretation. In branching time, } \Diamond \text{ is interpreted as "inevitability" – at some future point in every run.} \]
epistemic modalities $K_p$ and $P_p$ are "$p$ knows", and "in a state indistinguishable to $p$".

2.3.2 Semantics of $\mathcal{L}_{async}$

Kripke model semantics [Kri63] of modal logics requires an accessibility relation on the basic semantic entities of the model. For $\mathcal{L}_{async}$, the temporal operators' accessibility relation is $\preceq$ - causality between consistent cuts. For the epistemic operators we take the following definitions from [Tay90].

**Definition** [Taylor] Cuts $c = (h_{p_1}, h_{p_2}, \ldots, h_{p_n})$ and $c' = (h'_{p_1}, h'_{p_2}, \ldots, h'_{p_n})$ are indistinguishable to process $p_i$ (written $c \approx_{p_i} c'$) exactly when $h_{p_i} = h'_{p_i}$.

Formulae are evaluated along consistent cuts of an asynchronous execution. An interpretation for an infinite asynchronous run $A$, $\mathcal{I}_A$, assigns to each primitive proposition a subset of consistent cuts of $A$:

$$\mathcal{I}_A : \{a_1, a_2, \ldots\} \rightarrow 2^A.$$

We write

$$(A, c) \models a_i \text{ exactly when } c \in \mathcal{I}_A(a_i),$$

but will omit the run, and simply write $c \models \phi$, when it is clear from context. A consistent cut $c$ satisfies a well-formed formula, $\psi$, (written $c \models \psi$) according to the following structural rules:

- $\psi : \neg\phi$ iff $c \not\models \phi$;

- $\psi : \phi_1 \land \phi_2$ iff $c \models \phi_1$ and $c \models \phi_2$;
• \( \psi : \square \phi \iff \text{forall} \ c' \geq c. (c' \models \phi) \);

• \( \psi : \Box \phi \iff \text{forall} \ c' \leq c. (c \models \phi) \);

• \( \psi : \Diamond \phi \iff \text{forall} \ c' < c. (c \models \phi) \);

• \( \psi : \Diamond \phi \iff \text{in all runs for which} \ c \text{ is a prefix} \exists c' \geq c. (c' \models \phi) \).

• \( \psi : \Diamond \phi \iff \text{in all runs for which} \ c \text{ is a prefix} \exists c' > c. (c' \models \phi) \).

• \( \psi : P_p \phi \iff \text{exists} \ c' \in A, c' \approx_p c. ((A,c') \models \phi) \);

• \( \psi : K_p \phi \iff \text{forall} \ c' \approx_p c. (c' \models \phi) \);

2.3.3 Inference Rules and Axioms

Let \( \phi \) and \( \psi \) be well-formed formulae of \( \mathcal{L}_{async} \). The Inference Rules of \( \mathcal{L}_{async} \) are:

**Modus Ponens** \( \phi, \phi \Rightarrow \psi \vdash \psi \).

**Temporal Generalization** \( \models \Box \phi, \text{ and } \models \Box \phi. \)

**Epistemic Generalization** \( \phi \vdash K_p \phi. \)

The axioms of \( \mathcal{L}_{async} \) begin with all substitutional instances of truth-functional tautologies. Let \( \phi \) and \( \psi \) be well-formed formulae. The temporal axioms are:
\( T0a \quad \Box(\phi \Rightarrow \psi) \Rightarrow (\Box \phi \Rightarrow \Box \psi) \quad T0b \quad \phi \Rightarrow \Box \Diamond \phi \)

\( T0c \quad \Box(\phi \Rightarrow \psi) \Rightarrow (\Box \phi \Rightarrow \Box \psi) \quad T0d \quad \phi \Rightarrow \Box \Diamond \phi \)

\( T1a \quad \Box \phi \Rightarrow \Box \Box \phi \quad T1b \quad \Box \phi \Rightarrow \Box \Box \phi \)

\( T2a \quad \Box \phi \Rightarrow \phi \quad T2b \quad \Box \phi \Rightarrow \phi \)

\( T3 \quad \Box \bot \lor \Box \Box \bot \)

\( T4 \quad \Box \phi \Rightarrow \Diamond \phi \)

\( T5a \quad (\Diamond \phi \land \Diamond \psi) \Rightarrow \Diamond (\Diamond \phi \land \Diamond \psi) \quad T5b \quad (\Diamond \phi \land \Diamond \psi) \Rightarrow \Diamond (\Box \Diamond \phi \land \Box \Diamond \psi) \)

\( T6a \quad \Box \phi \Rightarrow \Box \phi \quad T6b \quad \Diamond \phi \Rightarrow \Diamond \phi \)

The epistemic axioms are from [Tay90]:

\( K0 \quad K_p \phi \Rightarrow \phi \quad K1 \quad K_p(\phi \Rightarrow \psi) \Rightarrow (K_p \phi \Rightarrow K_p \psi) \)

\( K2 \quad K_p \phi \Rightarrow K_p K_p \phi \quad K3 \quad \neg K_p \phi \Rightarrow K_p \neg K_p \phi \)

\( K4 \quad \phi \Rightarrow P_p \phi \quad K5 \quad P_p P_p \phi \Rightarrow P_p \phi \)

\( K6 \quad P_p \neg P_p \neg \phi \Rightarrow P_p \phi \quad K7 \quad P_p \phi \Rightarrow \neg K_p \neg \phi \)
Chapter 3

Strong Group Membership

In this chapter, we define formally the Strong Group Membership Problem for asynchronous systems. The definition should specify how to coordinate local events among a group of processes so that the externally observable behavior of the group (that is, the behavior observed by a process not in the group) is that of a single, fault-tolerant process. The ultimate goal is for any system process to be able to query core MRM members’ local views and observe “one-copy” [BH87] behavior on the sequence of view reports so-obtained. Because responses to queries will be taken as reflecting the exact system composition, we will want to ensure that MRM core members have identical sequences of view transitions. Failed core members will observe only a prefix of all view transitions, but their local views when they are operational must not be permitted to diverge. More important, core members that have been ousted but are not crashed, must not be permitted to misrepresent the system state. For example, we wish to preclude executions in which a functioning
process that is no longer a core member is able to change its local view independently; if that process were queried, its response could differ from the response a true core member (i.e. one that has been forced to coordinate changes to its local view) would give.

3.1 Formal Specification

To specify the problem formally, we define the following events, logical formulae, and sets of processes. For clarity, we will always use *emphasized* text for events, *small caps* for logical formulae, and *sans serif* for process sets.

The formula $\text{UP}_p$ holds along a cut $c$ if and only if $p$ has not executed $\text{quit}_p$ in its local history, $h_p$. Conversely, $\text{DOWN}_p$ holds along $c$ exactly when $p$ has crashed in $c$.\(^1\) The indexical set $\text{Up}(c)$ in an asynchronous run $A$ is the set of all processes that have not crashed along $c$:

$$\text{Up}(c) = \{ p \mid c \vdash \text{UP}_p \}.$$  

Process $p$ executes the internal event $\text{faulty}_p(q)$ as soon as it suspects $q$ faulty; whether $p$ comes to suspect $q$ through some local observation or through gossip is immaterial. Some time after recording $\text{faulty}_p(q)$, $p$ will execute the event $\text{remove}_p(q)$. Notice the distinction between the events $\text{faulty}_p(q)$ and $\text{remove}_p(q)$; the former represents $p$'s belief in $q$'s faultiness, which may be incorrect, while the latter is actual removal of $q$ from the set of core members $p$ believes operational. The formula $\text{FAULTY}_p(q)$ holds along all cuts that include $\text{faulty}_p(q)$, and

\(^1\)By definition, $\text{quit}_p$ will be the last event of $p$'s history in the cut.
REMOVE\(_p(q)\) along all cuts that include remove\(_p(q)\). Analogous statements hold for events operating\(_p(q)\) and add\(_p(q)\), and formulae OPERATING\(_p(q)\) ADD\(_p(q)\).

The local membership view for process \(p\) cut \(c = (h_1, \ldots, h_p, \ldots, h_n)\), (denoted \(\text{LocalView}_p(c)\)), 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 \(\text{LocalView}_p\) when the cut is clear from context. Trivially, we require \(p \in \text{LocalView}_p(c)\). The formula \(\text{IN-LOCAL}_p(q)\) holds along all cuts, \(c\), such that \(q \in \text{LocalView}_p(c)\). When \(\text{DOWN}_p\) holds along \(c\), \(\text{LocalView}_p(c)\) is undefined. Because \(h_p\) is linear, it makes sense to talk about the \(x^{th}\) version of \(p\)'s local view, which we denote \(\text{LocalView}_p^x\). Finally \(\text{IN-LOCAL}_p^x(q)\) holds when \(q \in \text{LocalView}_p^x\).

We extend local views to group views as follows. Given \(S \subseteq \text{Proc}\), 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, the group view is empty; and 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, \(S \subseteq \text{Proc}\), the group view determined by \(S\) along \(c\) is:

\[
\text{GpView}_S(c) = \begin{cases} 
\emptyset & S \cap \text{Up}(c) = \emptyset \\
\text{LocalView}_p(c) & (p, q \in S \cap \text{Up}(c)) \\
\text{LocalView}_p(c) = \text{LocalView}_q(c) & \text{otherwise.}
\end{cases}
\]
Lastly, the formula \( \text{IN-GP}_p \) holds along all cuts, \( c \), such that \( p \in \text{GpView}_S(c) \); \( \text{OUT-GP}_p \) holds when \( p \notin \text{GpView}_S(c) \). Observe that precise definition of this formula anticipates uniqueness of the group view along all consistent cuts.

### 3.1.1 More About \( \text{GpView}_S(c) \)

The definition of \( \text{GpView}_S(c) \) is crucial to the class of Group Membership Problems, so it is worthwhile discussing a subtle point: how the sets \( S \) and \( \text{GpView}_S(c) \) relate. Recall that \( \text{GpView}_S(c) \) is the abstraction we are using to define the ultimate goal—the single, fault-tolerant process illusion we will use to build our MRM. In this light, MRM core members are precisely the members of \( \text{GpView}_S(c) \), so any member of \( \text{GpView}_S(c) \) should be able to reply to MRM client requests.

Now if \( q \in \left( \text{GpView}_S(c) \cap \bar{S} \right) \) then \( q \) is a core member whose local view is not used in determining the MRM composition and system state; specifically, \( q \)'s local view is not constrained by the definition of \( \text{GpView}_S(c) \), so \( \text{LocalView}_q(c) \) need not be identical to \( \text{GpView}_S(c) \).² Because \( q \) replies to MRM client requests based on its local view, its replies will contradict other core members' replies when \( \text{LocalView}_q(c) \neq \text{GpView}_S(c) \). Thus, the single-process illusion falls apart, and the MRM may easily fall short of its stated aim in assuring global consistency, when every core member's local view is not accounted for in determining the MRM group view.

To avoid this, we need a clause in the specification requiring \( q \) to be in \( S \) when-

²In practice, a core member's local view includes in addition to the MRM composition, the entire system process composition.
ever it is in $\text{GpView}_S(c)$:

$$\left( \text{GpView}_S(c) \cap \text{Up}(c) \right) \subseteq \left( S \cap \text{Up}(c) \right).$$

Since $p \in \text{LocalView}_p(c)$,

$$\left( S \cap \text{Up}(c) \right) \subseteq \left( \text{GpView}_S(c) \cap \text{Up}(c) \right),$$

and so the desired clause will require $S = \text{GpView}_S(c)$.

Note, too, that the MRM must be unique to maintain the single-process illusion. The desired clause will incorporate the above-mentioned equality as well as uniqueness along all consistent cuts of the set, $S$, satisfying equality.

### 3.1.2 Strong GMP Specification

We now have the language and formal tools necessary to formalize Strong GMP.

**GMP-0** (Base Case) An initial group view exists along the initial cut:

$$\bigvee_{S_0 \subseteq \text{Proc}} \left( S_0 = \text{GpView}_{c_0}(S_0) \right).$$

**GMP-1** (Validity) Processes do not make changes to their local views capriciously:

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

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

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.
For all cuts, \( c \):

\[
\bigvee_{S \subseteq \text{Proc}} \left( \text{GpView}_S(c) = S \right) \Rightarrow \bigwedge_{\emptyset \neq S' \neq S} \text{UNDEF'}(\text{GpView}_{S'}(c))
\]
GMP-3 (Sequence) All processes exhibit the same sequence of local views contemporaneously, provided the views are defined:

\[ \bigwedge_{0 \leq x} \bigwedge_{p} \bigwedge_{q} \left( \neg \text{IN-LOCAL}^{x}_{p}(q) \Rightarrow \text{UNDEF-D}(\text{LocalView}^{x}_{p}) \right) \land \\
\left( \text{IN-LOCAL}^{x}_{p}(q) \Rightarrow \text{DOWN}_{q} \lor \left( \text{LocalView}_{q} = \text{LocalView}^{x}_{p} \right) \right). \]

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

a. \( \text{FAULTY}_{p}(q) \land \text{IN-GP}_{p} \Rightarrow \left( \Diamond \text{OUT-GP}_{q} \lor \Diamond \text{OUT-GP}_{p} \right) \)

b. \( \text{OPERATING}_{p}(q) \land \text{IN-GP}_{p} \Rightarrow \left( \Diamond \text{IN-GP}_{q} \lor \Diamond \text{OUT-GP}_{p} \right) \).

GMP-3 is equivalent to requiring that each local view eventually become a group view. The presence of \( \Diamond \) forces a group view to exist along some consistent cut in an execution. This too, is why we cannot bind \( \text{LocalView}_{q} \) to a version number. Local views when indexed by version numbers are static – the composition of a process’s \( x^{th} \) local view will never change. So suppose \( c \) is the witness for \( \Diamond \). Then omitting the version superscript forces \( \text{LocalView}_{q}(c) \) (for \( q \in \text{LocalView}^{x}_{p} \)) to be identical to \( \text{LocalView}^{x}_{p} \) 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 \( \ll \)-related. Hence, it makes sense to talk about the \( x^{th} \) group view, which we denote \( G_{pView}^x \).
Chapter 4

A Protocol Solving Strong GMP

This chapter presents a distributed protocol that solves Strong GMP and therefore achieves the single, fault-tolerant process illusion we discussed in Section 1.2.2.1. The Strong Group Membership Protocol (hereafter S-GMP) is both asymmetric and centralized: a distinguished core member, denoted $mgr$, is responsible for coordinating updates to the outer (i.e. non-$mgr$) core members’ local views. In a symmetric, distributed solution all core members would behave identically and make updates independently. We chose the centralized approach for two main reasons: it requires only $O(n)$ point-to-point messages,\(^1\) instead of $O(n^2)$, and it is a simpler paradigm within which to reason. While the coordinator’s failure is more troublesome to handle than an outer member’s, the benefits of the centralized approach, coupled with the extremely low probability of the coordinator failing outweigh these concerns.

In large systems, we would solve Strong GMP with a hierarchically-structured

\(^1\)At most $3(n - 2)$, and when amortized over successive updates, approximately $n$ per update.
service. In this design, the protocol discussed in this chapter would be run by a small cadre of processes, which we have already termed 'the core'(1.2.2.1). The core would then use a cheap replication scheme (e.g. the Isis replication tools) to maintain a fault-tolerant member list for the overall system.

We develop the protocol in stages, first giving a simplified version of S-GMP for the case when mgr does not fail, and then discussing the more complicated problem of reconfiguration. To introduce S-GMP we discuss only core member removals, but analogous techniques will be used when new members join. Chapter 5 proves S-GMP correct for both removals and joins. We use \( <v, x> \) to denote an update to \( \text{GpView}^e \), in which \( v \) (the value proposed) is a set of process identifiers. When the version number is clear from context we simply write \( v \). We will also sometimes want to identify the specific processes in \( v \).

4.1 Simple S-GMP

While we assume in this section that mgr does not fail, the protocol we present is in fact more complicated in communication structure and degree of coordination than this assumption warrants. Indeed, if we knew that mgr could not fail we would already have a single, fault-tolerant process, and we could solve Strong GMP with a reliable broadcast. Instead, we anticipate mgr’s failure to simplify presenting the reconfiguration algorithm in Section 4.2.

When mgr becomes aware of an outer member’s, say r’s, failure, it initiates a two-phase update algorithm. In Phase 1 (Figure 4.1) mgr proposes r’s removal by
multicasting\(^2\) a *submit* message, \texttt{M-sub}(\(-r\)), to the members of its local view and awaits each outer core member’s response or its own belief in an outer member’s faultiness. In this way all core members that are not believed faulty by \texttt{mgr} believe \(r\) faulty at the end of Phase I. If \texttt{mgr} receives responses from a majority of its current local view, it multicasts a *commit* message, \texttt{M-com}(\(-r\)), in Phase II.\(^3\) If \texttt{mgr} does not receive a majority response, it must block. If local views are identical at the outset of Simple S-GMP then, because \texttt{mgr} is a single process, local views are identical at the end of Simple S-GMP.

The submit message coordinates belief among the group in \(r\)’s faultiness; the commit message tells outer members that the group has reached *agreement* on \(r\)’s

\(^2\)Multicasts are not failure atomic.

\(^3\)Typically, a phase of communication consists of a multicast from a single process to a group of processes and their responses back to the message initiator. In fact, Simple S-GMP is one-and-one-half phases, but this is awkward.
failure and that they should now remove $r$ from their own local views. This agreement, however, need not be complete: because $mgr$ does not receive responses from outer members it believes faulty, it does not know whether these members received its submit message. Even though such a member may receive $mgr$’s messages, $mgr$ certainly has no way of knowing this. From $mgr$’s perspective, these members may not be aware of the current update to the group view, rendering complete, core-wide agreement on the new view contingent upon the subsequent removal of these faulty core members. Finally, Gossip ensures that operational outer processes become aware of such contingencies.

Observe that the Phase I submit message is unnecessary if $mgr$ knows a major-
ity of the non-faulty outer processes already believe a process, say \( q \), faulty. That is, \( mgr \) need not disseminate its own belief in \( q \)'s faultiness if it knows that the outer members already believe the same thing. In this light, the contingent updates when piggy-backed upon a commit message, 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 Phase II commit message, exactly how it plans to change the group view next. For example, in Figure 4.2, process \( q \) crashes before replying to \( M\text{-sub}(−r) \) causing \( mgr \) to suspect \( q \) faulty. By appending \( M\text{-sub}(−q) \) to its commit message \( M\text{-com}(−r) \), \( mgr \) indicates that it will next attempt to remove \( q \) from the group view. Outer processes respond to the combined commit-submit message as they do to plain submit messages.

4.2 Full S-GMP

When \( mgr \) fails (or is believed to have failed) the outer members execute a reconfiguration algorithm to select a new coordinator and, if necessary, reestablish the group view. To see how this may arise, consider Figure 4.3 in which \( mgr \) fails in the middle of a commit multicast; because local views differ along the second cut, the group view is undefined.

Reconfiguring safely and 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 members’ inconsistencies and maintain safety?

In practical terms, a correct reconfiguration depends on an initiator’s ability to
Figure 4.3: \textit{mgr}'s Failure Results in Undefined Group View.

determine the last defined group view and propagate the correct submission for the succeeding group view. As Figure 4.3 shows, proposals may also be partially known among the current group view, and this can pose problems in reconfiguration.

The most intriguing and difficult aspect of reconfiguration is accounting for \textit{invisible commits}. An invisible commit is an update that is committed by a set of processes, but can never be known to have been committed by them. An invisible commit occurs when the only processes receiving a commit message fail, or are believed faulty by the rest of the group. This is significant for reconfiguration: while no subsequent reconfiguration initiator will ever \textit{know} whether these processes received the commit messages and updated their local views, Strong GMP (specifically, GMP-3) requires that if an invisible commit did occur, the remaining core members must behave in a manner consistent with that event. It is imperative, then, that every invisibly committed update be detectable by every reconfigurer.

This is the core issue in maintaining GMP-3. We can ensure this only if all initia-
tors (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 Structure of the Reconfiguration Algorithm

Unlike the mgr-initiated algorithm, reconfiguration sometime requires three phases. In actuality three phases are necessary only in the worst and highly unlikely case, but for simplicity we present the algorithm here as three phases and discuss the cases when two suffice in Chapter 6. That reconfiguring is three-phase is interesting and important, though not surprising in light of Skeen's work on non-blocking commit protocols [Ske82]. In the first phase, the initiator $r$, broadcasts a reconfiguration interrogate message, $\textsf{R-int}(\text{ver}(r))$, to all processes in its local view and awaits their responses or its own belief in their faultiness. Core members respond to the interrogate message with their current, local state. If a majority have responded, the initiator uses the information it received to determine an update event, say $v$, and version number, say $x$, whose execution would result in a well-defined $\textsf{GpView}^r$. The initiator multicasts this event as the Phase II reconfiguration submit message, $\textsf{R-sub}(<v, x>)$. After obtaining a second majority response, acknowledging receipt of the submission, the initiator broadcasts the Phase III reconfiguration commit message, $\textsf{R-com}(<v, x>)$. Again, majority approval of $\textsf{R-int}(\text{ver}(r))$ and $\textsf{R-sub}(<v, x>)$ are essential in maintaining Sequence and Uniqueness; without either, the initiator must block.

**Definition** An update initiator (either mgr or a reconfigurer) is *successful* for its submission ($\textsf{M-sub}(<v, x>)$ or $\textsf{R-sub}(<v, x>)$) if it can reach the appropriate
s-GMP commit phase; that is, if a majority subset of its local view respond to the proposal.

A submitted value, $<v, x>$, is stable if and only if the initiator submitting it is successful. ■

In this light, GMP-3 requires that all successful initiators trying to install (or complete the installation of) the $x^{th}$ system view make identical proposals. The local state information collected during reconfiguration Phase 1 must allow the reconfiguration initiator 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, and propose \( mgr \)'s removal if they do not. Unfortunately, as Figure 4.1 makes clear, asynchrony and inopportune failures can result in there being two different proposals for the same instance of the group view. Correctness requires that only one of them become stable, and that any non-blocking reconfigurer be able to determine which one it is by the end of reconfiguration Phase I. 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.\(^4\) We use \( \text{rank}(p) > \text{rank}(q) \) to denote that \( p \) has higher rank than \( q \). Whenever a process is removed from the group view, the ranks of all higher-ranked processes are decreased by one. Observe that for as long as \( p \) and \( q \) are both (contemporaneously) core members, their ranking *relative to each other* will not change.

A process initiates reconfiguration when it believes all those ranked higher than itself are faulty. That is, given cut \( c \) and \( \text{LocalView}_p(c) \), define the local proposition

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

\(^4\)This does not mean we cannot join multiple processes to the group view in a single update.
<table>
<thead>
<tr>
<th>Process rank</th>
<th>UP (_p)</th>
<th>FAULTY(_q(p))</th>
<th>INITIATE(_q)</th>
<th>INITIATE(_p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank((mgr) = \rho)</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>rank((p) = \rho - 1)</td>
<td>False</td>
<td>False</td>
<td>Eventually</td>
<td>False</td>
</tr>
<tr>
<td>rank((q) = \rho - 2)</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

While initiating reconfiguration on INITIATE\(_p\) can lead to multiple, concurrent reconfiguration initiations, it guarantees at least one process will begin the reconfiguration algorithm. Consider Table 4.2.2 in which rank\((mgr) = \rho\), rank\((p) = \rho - 1\), and rank\((q) = \rho - 2\), and both \(p\) and \(q\) believe \(mgr\) faulty. In the third scenario both \(p\) and \(q\) initiate reconfigurations. S-GMP must ensure view uniqueness in the face of multiple, concurrent reconfiguration attempts. In the second scenario \(q\) expects \(p\), which has crashed, to initiate a reconfiguration. Any solution must also ensure that \(q\) eventually comes to suspect \(p\) faulty. In S-GMP, \(q\) times-out waiting for \(p\)'s R-int() message, surmises FAULTY\(_q(p)\), and then initiates reconfiguration.

### 4.2.3 Rules of Progression: Maintaining Safety

To understand the difficulties encountered in reconfiguration we examine GMP-2 and GMP-3 more closely.

Uniqueness requires that at most one group view exists along any consistent cut in an asynchronous execution. In the situation depicted in Figure 4.5, \(Q\) and \(R\) are subsets of GpView\(^x\), and \(q\) and \(r\) are both initiating reconfiguration. If all members
of $Q$ believe $r$ faulty, they will receive neither $r$'s R-int() message, nor its Phase II proposal, nor its Phase III commit. 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 $Q$. Since $Rcup\{r\}$ will eventually remove all of $Qcup\{q\}$, and $Qcup\{q\}$ will eventually remove all of $Rcup\{r\}$, two distinct group views will exist.

Naively, it would appear that the majority requirement suffices to preclude this occurring. However, as Figure 4.3 makes clear, initiators that may end up installing (submitting and committing) the same group version need not have identical local views when they undertake reconfiguration. Any solution must ensure GMP-2 when concurrent initiators are seeking majority approval from different sets of processes.
Henceforth, let $\mu_r(c)$ be the size of the smallest majority subset of $\text{LocalView}_r(c)$, and $\mu^r_r$ be the size of the smallest majority subset of $\text{LocalView}^r_r$:

$$\mu_r(c) \overset{\text{def}}{=} \left\lfloor \frac{|\text{LocalView}_r(c)|}{2} \right\rfloor + 1, \quad \mu^r_r \overset{\text{def}}{=} \left\lfloor \frac{|\text{LocalView}^r_r|}{2} \right\rfloor + 1.$$ 

### 4.2.4 Reconfiguration Phase I Responses

Outer processes' responses to $R\cdot\text{int}(\text{ver}(r))$ must allow $r$ to determine the nature and composition of all local view inconsistencies, including inconsistencies involving core members that did not respond to $r$. While local views alone suffice to detect inconsistencies among the processes responding to an initiator, this information falls short of satisfying GMP-3 (Sequence) entirely as invisible commits are not detectable.
In Figure 4.6 $< v, x >$ 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.

4.2.4.1 Remarks

Since fault determinations are permanent (c.f. Section 2.1) all processes, but particularly initiators, should make them cautiously. Any core member believing a majority of its local view faulty can never succeed in committing an update. Moreover, Gossip ensures that future communication with another core member results in the latter also believing this majority faulty. Recovering from such a complete failure is difficult; one could employ, for example, Skeen’s algorithm [Ske85], but a complete discussion of recovering from total failure is beyond the scope of this work.

Before presenting S-GMP we should address issues arising at start up. In practice, a process wanting to join the MRM will query the system name service for the MRM. If the name service reports that the MRM is non-existent, this process executes an initiation sequence and becomes the MRM. If the name service reports that the MRM already exists, the process requests of the MRM that it be allowed
to join. The MRM core must build to *critical size*, after which it begins operating as a system service, and until that point, the name service is not fault-tolerant: it is either a single process that may fail, or an uncoordinated group of processes that may report inconsistent information regarding the MRM. In the latter case, a race condition may occur if more than one process desiring to join the MRM is told it does not exist.

4.3 The S-GMP Algorithm

The most important, and rather astonishing, aspect of our algorithm is the complete lack of restrictions on changes to a group view. Specifically, there is no upper limit on the number of processes (i.e. core members) 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 has a number of very important ramifications. Compared to an MRM defined by a less-stringent member of the class of Asynchronous Group Membership Problems (we make this explicit in Chapter 7), an MRM defined by Strong GMP has a number of striking advantages:

**Fault-tolerance** The Strong GMP service can tolerate nearly half its members failing simultaneously. Moreover, the fault-tolerance level is dynamic and depends only on the size of the current group view. Weaker versions can tolerate the simultaneous failure of only a fixed number of members, and given initial conditions, this will usually be one or two.
Responsiveness The Strong GMP service can adapt much more quickly to severe changes in its work load. In a single change to its core membership, the Strong GMP service can add an arbitrary number of new members to share a suddenly-increased load, or remove nearly half its members when its work load drops quickly (thereby ridding itself of 'dead weight'). Weaker protocols will need to run as many as \( n \) rounds to change membership by \( n \) processes.

Additions require very little change to S-GMP as presented in the first two sections of this chapter. Let \( v_1 \) be the set of new members. An initiator, say \( r \), sends a Join message, giving the members of \( v_1 \) permission to join, immediately before it multicasts the commit message, \( \text{com}(<v_1, x + 1>) ; \text{M-sub}(<v', x + 2>) \), to \( \text{GpView}^z \). The initiator also sends a State-Xfer message to \( v_1 \), informing the new members of all relevant core and system state. To simplify bookkeeping, new members begin with local version equal to the group version their addition resulted in.

If \( p' \in v_1 \) had not already received the state transfer information from a previous initiator it initializes its local state and sends \text{ack}(\text{Join}) \) back to \( r \). If \( p' \) has already joined at the behest of a previous initiator, say \( r_0 \), \( p' \) responds to \( r \) with \text{NextUpdate}_{p'} \) as set by \( r_0 \). This response is necessary to maintain GMP-2 and GMP-3; it deals with the case in which \( r_0 \)'s commit message differed from \( r \)'s in the contingent submission component, say \( \text{M-sub}(<v_2, x + 2>) \), and \( r_0 \) may have been able to commit \( <v_2, x + 2> \) invisibly to \( r \) and all members of \( \text{Acks}(r, \text{R-int}(\text{ver}(r))) \) (See Figure 4.3).
Figure 4.7: A Situation Requiring New Core Members to Report \textbf{NextUpdate} to Initiator.
In what follows \(v1\) and \(v2\) are sets of process identifiers. For full generality we use \(\pm\) to denote either of the operations that alter local views - \(\text{remove}()\) or \(\text{add}()\). Given a set of processes, \(G\), let \(\text{multicast}_p(G, m)\) denote the compound action \(\forall q \in G: (\text{send}_p(q, m))\). \(\text{multicast}_p(G, m)\) 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 \(\text{ack}(m)\) acknowledges receipt of message \(m\).

The sets \(\text{Faulty}_p\) and \(\text{Recovered}_p\) are, respectively, the members of its local view \(p\) believes faulty, and the members not in its local view \(p\) currently believes functional.

For simplicity we do not explicitly show gossiping, channel-disconnect, or any error-checking, but assume these are done transparently. For example, we do not show an outer process checking whether the process from which it has received an \(\text{M-sub}()\) message is, in its belief, the current \(\text{mgr}\), whether the update indicated in a \(\text{com}()\) message actually matches the update indicated in the most recent \(\text{sub}()\) message, whether the processes to be removed (or added) are currently (not) in the group view, and so forth.
Task: \textit{mgr}

while (true)

    repeat

    GetUpdate(v1);

    until (v1 ! = nil-id);

    \textit{multicast}_{mgr} (\textit{LocalView}_{mgr}, M-sub(\pm v1));

    while (v1 ! = nil-id) /* Compressed algorithm loop. */

    forall \( p \in \textit{LocalView}_{mgr} \)

    recv_{mgr} (p, \textit{ack}(M-sub(\pm v1))) or faulty_{mgr} (p);

    if (majority of \textit{LocalView}_{mgr} didn’t respond) quit_{mgr};

    /* Update \textit{LocalView}_{mgr} according to \pm. */

    DoCommit(v1);

    GetUpdate(v2);

    if (Joining new members)

    \textit{multicast}_{mgr} (v1, Join : State-Xfer);

    forall \( p' \in v1 \)

    recv_{mgr} (v1, \textit{ack}(Join) : \textit{NextUpdate}_{p'}) or faulty_{mgr} (p');

    \( v2 = \textit{NextUpdate}_{v1}; \)

    \textit{multicast}_{mgr} (\textit{LocalView}_{mgr}, M-com(\pm v1) : M-sub(\pm v2));

    v1 = v2;
/* Called only by mgr. */

Procedure: GetUpdate(OUT val);

if (Recovered\text{mgr} \neq \emptyset) val = Recovered\text{mgr} ;

else val = Faulty\text{mgr} ;

return();

Task: Outer Processes, p

\text{recv}_p(mgr, M-sub(\pm v1));

/* Mark the processes of v1 faulty or operational as appropriate. */
DoPreCommit(v1, \pm);

repeat

\text{send}_p(mgr, \text{ack}(M-sub(\pm v1)));

\text{recv}_p(mgr, M-com(\pm v1) : M-sub(\pm v2)) or faultyp(mgr);

if (\neg FAULTY_p(mgr))

DoPreCommit(v2);

DoCommit(v1);

v1 = v2;

else Wait-Reconfiguration();

until (v1 == nil-id);
/* Executed by a process destined to be an MRM member. */

New-Process: \( p' \)

name-server-Lookup(MRM, contact-id);

if (contact-id == nil-id)
    /* Race Conditions Possible. */
    \( \text{LocalView}_{p'}^1 = \{ p' \} \);
    name-server-Install(MRM, \( p' \));
    Begin \( mgr \) Task;
else
    repeat
        \( \text{send}_{p'}(\text{contact-id, request-join-MRM}); \)
        \( \text{recv}_{p'}(\text{contact-id, Join( < v, x > ) : State-Xfer} \) OR time-out;
        if (time-out)
            name-server-Lookup(MRM, contact-id);
        else /* \( \text{LocalView}_{p'} \) is part of \( \text{State-Xfer} \). */
            DoCommit(< v, x >);
    until (MRM-member OR MRM-unavailable);
Reconfiguration

For Reconfiguring, we introduce the following variables:

- **NextUpdate**$_p$ is a tuple of the form $[lds, \text{ver}(p) + 1, \text{rank}(\text{init})]_p$, where $lds$ is the set of processes by which $p$ is expecting to alter its current local view, $\text{LocalView}_p^{\text{ver}(p)}$, and $\text{rank}(\text{init})$ is the rank (in $\text{LocalView}_p^{\text{ver}(p)}$) of the process from which $p$ received the proposal. When $p$ receives a submission it changes $\text{NextUpdate}_p$ to reflect the value proposed and the initiator proposing it.

- **LastCommit**$_p$ is a tuple of the form $[lds, \text{ver}(p)]_p$, where $lds$ is the set of processes by which $p$ altered its previous local view to obtain its current local view $\text{LocalView}_p^{\text{ver}(p)}$.

- **Ahead**$_r$ is the set values reported committed for versions numbered greater than $\text{ver}(r)$. Initiator $r$ receives these values in response to its $\text{R-int}(\text{ver}(r))$ message. **Behind**$_r$ are reported local versions numbered less than $\text{ver}(r)$. Anticipating the correctness proofs of Chapter 5, the reported version in **Ahead**$_r$ is exactly $\text{ver}(r) + 1$, and that in **Behind**$_r$ is exactly $\text{ver}(r) - 1$.

- **SubCurrent**$_r$ is the set of proposed next updates with proposed versions equal to $\text{ver}(r) + 1$; **SubAhead**$_r$ is the set with proposed versions greater than $\text{ver}(r) + 1$. 

Task: Reconfiguration Initiator, \( r \), with \( \text{ver}(r) = x \)

\( \text{multicast}_r(\text{LocalView}_r, R-\text{int}(x)) ; \)

for all \( p \in \text{LocalView}_r \)

\( \text{recv}_r(p, \text{NextUpdate}_p : \text{LastCommit}_p) \) or \( \text{faulty}_r(p) ; \)

if (majority of \( \text{LocalView}_r \) didn’t respond) \( \text{quit}_r ; \)

/* Determine the value and version to submit from the responses received. */

DetermineProposal(\( v_1, \text{ver}, v_2 \));

DoPreCommit(\( v_1 \));

\( \text{multicast}_r(\text{LocalView}_r, R-\text{sub}(\langle \pm v_1, \text{ver} \rangle)) ; \)

for all \( p \in \text{LocalView}_r \)

\( \text{recv}_r(p, \text{ack}(R-\text{sub}(\langle \pm v_1, \text{ver} \rangle))) \) or \( \text{faulty}_r(p) ; \)

if (majority of \( \text{LocalView}_r \) didn’t respond) \( \text{quit}_r ; \)

DoCommit(\( v_1 \));

if (Joining new members)

\( \text{multicast}_r(v_1, \text{Join} : \text{State-Xfer}) ; \)

for all \( p' \in v_1 \)

\( \text{recv}_r(v_1, \text{ack}(\text{Join}) : \text{NextUpdate}_{p'}) \) or \( \text{faulty}_r(p') ; \)

\( v_2 = \text{NextUpdate}_{v_1} ; \)

\( \text{multicast}_r(\text{LocalView}_r, R-\text{com}(\langle \pm v_1, \text{ver} \rangle) : R-\text{sub}(\pm v_2)) ; \)

\( \text{mgr} \cdot v_1 = r, v_2 ; \)

Begin \( \text{mgr} \) Task;
Task: Outer Reconfiguration, p

\( \text{recv}_p(r, R\text{-int}(x)) ; \)

\(/ * \text{If rank}(p) > \text{rank}(r), \text{I'm thought faulty.} */\)

\( \text{send}_p(r, \text{NextUpdate}_p : \text{LastCommit}_p) ; \)

\( \text{recv}_p(r, R\text{-sub}(< \pm v1, ver >)) \text{ or fault}_p(r) ; \)

if (!\text{FAULTY}_p(r))

\( \text{DoPreCommit}(v1) ; \)

\( \text{send}_p(r, \text{ack}(R\text{-sub}(< \pm v1, ver >))) ; \)

\( \text{recv}_p(r, R\text{-com}(< \pm v1, ver >) : M\text{-sub}(\pm v2)) \text{ or fault}_p(r) ; \)

if (!\text{FAULTY}_p(r))

\( \text{DoCommit}(v1) ; \)

\( \text{mgr}, v1 = r, v2 ; \)

else Wait-Reconfiguration();

else Wait-Reconfiguration();
/* Sets proposal and invisible from among the */

/* NextUpdate and LastCommit information. */

/* ver(r) = x. */

Procedure: DetermineProposal(OUT proposal, OUT version, OUT invisible);

Ahead<sub>r</sub> = \{\{lds, ver(p)\} \mid ver(p) = (x + 1)\};

Behind<sub>r</sub> = \{\{lds, ver(p)\} \mid ver(p) = (x - 1)\};

SubAhead<sub>r</sub> = \{\{lds, ver(p) + 1, rank(init)\} \mid ver(p) = (x + 1)\};

SubCurrent<sub>r</sub> = \{\{lds, ver(p) + 1, rank(init)\} \mid ver(p) = x\};

if (Ahead<sub>r</sub> ≠ ∅)

/* Partially committed version x + 1. */

proposal = Ahead<sub>r</sub>;

GetStableProposal(invisible, SubAhead<sub>r</sub>);

version = x + 1;

return();

if (Behind<sub>r</sub> ≠ ∅)

/* Partially committed version x. */

proposal = \{lds, ver(r)\}; /* LastCommit<sub>r</sub> */

GetStableProposal(invisible, SubCurrent<sub>r</sub>);

version = x;

return();
DetermineProposal, continued.

/** All respondents report the same local version. */

version = x + 1;

if (SubCurrent_r == ∅)
   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();
Chapter 5

Correctness

In this chapter we prove $s$-GMP correctly solves Strong GMP. We prove correctness by induction. In the first section of this chapter, we construct the inductive step; full correctness follows easily from there.

5.1 The Inductive Step

Throughout this chapter we use $\text{sub}()$ and $\text{com}()$ to denote submit and commit messages irrespective of the sender's role ($\text{mgr}$ or a reconfigurer). As in Section 4.3, $\text{NextUpdate}_p$ is the triple $[\text{lds}, \text{ver}(p) + 1, \text{rank}(init)]_p$. For each $p$, Gossip, Disconnect and INITIATE() mean that $\text{NextUpdate}_p$ is always the proposal of the lowest-ranked initiator from which $p$ has received proposed updates for group view $\text{ver}(p) + 1$.

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

\[
\text{Ahead}_r \overset{\text{def}}{=} \{ p \mid p \in \text{Acks}(r, \text{R-int}(\text{ver}(r))) \land \text{ver}(p) > \text{ver}(r) \}
\]

\[
\text{Behind}_r \overset{\text{def}}{=} \{ p \mid p \in \text{Acks}(r, \text{R-int}(\text{ver}(r))) \land \text{ver}(p) < \text{ver}(r) \}
\]

We also assume \( \text{GpView}^{x-1} \) is well-defined. We approach the inductive step by proving all successful initiators propose the same value for \( \text{GpView}^x \), which goes a long way toward maintaining both GMP-2 and GMP-3 as this is the only value that can be committed. The most difficult aspect is ensuring that reconfigurers are able to detect updates that are committed invisibly to them.

**Proposition 5.1.1** If \( r \) is a reconfiguration initiator with \( \text{ver}(r) = x \), then

\[
\bigwedge_{q \in \text{Acks}(r, \text{R-int}(x))} \left( x - 1 \leq \text{ver}(q) \leq x + 1 \right).
\]

**Proof** Though \( \text{GpView}^x \) may not be defined globally, let \( p \) be the process responsible for \( r \) installing \( \text{LocalView}_r^x \). Suppose some \( q \in \text{Acks}(r, \text{R-int}(x)) \) reports \( \text{ver}(q) < x - 1 \). Then \( q \) has neither received nor responded to \( p \)'s \( \text{sub}(<v, x>) \), so \( p \) believes \( \text{FAULTY}_p(q) \). Gossip ensures \( r \) also believes \( \text{FAULTY}_r(q) \) upon receipt of \( \text{com}(<v, x>) \) from \( p \). Since \( r \) received \( \text{com}(<v, x>) \) before it multicasted \( \text{R-int}(x) \), the Disconnect property makes it impossible for \( r \) to receive a response from \( q \). Thus, \( q \not\in \text{Acks}(r, \text{R-int}(x)) \).

So suppose some \( q' \in \text{Acks}(r, \text{R-int}(x)) \) reports \( \text{ver}(q') > x + 1 \), and let \( p' \) be responsible for \( q' \) installing \( \text{ver}(q') \). Because \( x \leq \text{ver}(q') - 2 \), \( r \) has neither received nor responded to \( p' \)'s \( \text{sub}(<v, \text{ver}(q')>) \), resulting in \( \text{FAULTY}_{p'}(r) \), and upon \( q' \)'s receipt of \( \text{com}(<v, \text{ver}(q')>) \), \( \text{FAULTY}_{q'}(r) \). Again, Disconnect prevents \( q' \) from
receiving and responding to $r$'s $R\text{-int}(x)$; $q' \notin \text{Acks}(r, R\text{-int}(x))$.

Definition Given process $p$, $\text{GpView}^x$ is $p$-defined (along consistent cut $c$) if and only if

$$\bigwedge_{q \in \text{LocalView}_p} \left(K_p \text{ver}(q) \geq x\right) \lor \text{FAULTY}_p(q). \quad (5.1)$$

That is, from $p$'s point of view, $\text{GpView}^x$ is (or has been) defined. Of course $\text{GpView}^x$ may not be defined globally as some process $q$ that $p$ believes faulty, may have $\text{ver}(q) < x$. With respect to a reconfiguration initiator, $r$, $\text{GpView}^x$ is $r$-defined at the end of Phase 1, if every process in $\text{Acks}(r, R\text{-int}(x)) - \text{Faulty}_r$ reported a local version at least as large as $x$. Processes in $\text{Acks}(r, R\text{-int}(x)) \cap \text{Faulty}_r$ may have reported local versions less than $x$, but $r$ believes them faulty by the end of Phase 1.

**Proposition 5.1.2** Let $r$ be a reconfiguration initiator. Then $r$ proposes version $x$ if and only if $\text{GpView}^{x-1}$ is the most recent (i.e. highest-numbered) $r$-defined system view at the end of Reconfiguration Phase 1.

Proof Refer to procedure $\text{DetermineProposal}$ in Section 4.3.

- When $\text{Ahead}_r \neq \emptyset$, $r$ proposes version number $\text{ver}(r) + 1$. While it may also be the case that $\text{Behind}_r \neq \emptyset$, we can use Proposition 5.1.1 to show that $\text{FAULTY}_p(q)$ holds for $p \in \text{Ahead}_r$ and $q \in \text{Behind}_r$ resulting in $\text{FAULTY}_r(q)$ at the end of Phase 1 and $\text{GpView}^{\text{ver}(r)}$ being $r$-defined there.
• When $\text{Ahead}_r = \emptyset$ and $\text{Behind}_r \neq \emptyset$, $\text{GpView}^{\text{ver}(q)}$ for $q \in \text{Behind}_r$ is the most recent $r$-defined system view; $r$ proposes version number $\text{ver}(r) = \text{ver}(q) + 1$.

• When $\text{Ahead}_r = \text{Behind}_r = \emptyset$, $\text{GpView}^{\text{ver}(r)}$ is the most recent $r$-defined system view and $r$ proposes version $\text{ver}(r) + 1$.

\[
\]

**Proposition 5.1.3** For any initiator, $p$, if $p$ submits $\text{sub}(<v, x>)$, then

\[
x - 1 \leq \text{ver}(p) \leq x.
\]

**Proof** By way of contradiction, suppose $x < \text{ver}(p)$. From Proposition 5.1.2, we know $\text{GpView}^{x-1}$ is the most recent $p$-defined group view, so let $q$ be any outer member reporting $\text{ver}(q) = x - 1$. Let $p'$ be the process from which $p$ received the commit message $\text{com}(<v', \text{ver}(p)>)$. Since $q$ has not committed view $x$, it can neither have received $\text{sub}(<v', \text{ver}(p)>)$ from $p'$, nor have responded back to $p'$. By the end of $p'$'s submit phase $\text{FAULTY}_{p'}(q)$ holds, and by Gossip so does $\text{FAULTY}_p(q)$ as soon as $p$ receives $\text{com}(<v', \text{ver}(p)>)$; $p$ would not have received $q$'s response to $\text{R-int}(\text{ver}(p))$.

On the other hand, $\text{ver}(p) < x - 1$ is impossible if $\text{GpView}^{x-1}$ is the most recent $p$-defined view.
Proposition 5.1.4 If $q$ with $ver(q) = x - 2$ receives $\text{sub}(<v,x>)$ from initiator $r$, then $\text{FAULTY}_r(q)$ holds when $r$ multicasted the submission:

$$\bigwedge_{q \in \text{LocalView}_r} \left( ver(q) = x - 2 \land \text{SEND}_r(q, \text{sub}(<v,x>)) \right) \Rightarrow \text{FAULTY}_r(q).$$

Proof Let $ver(q) = x - 2$ and suppose $r$ proposes $<v,x>$. It is not possible for $r$ to be the same co-ordinator as the one from which $q$ received $\text{com}(<v'',x - 2>)$ because FIFO channels force $q$ to receive $\text{sub}(<v',x - 1>)$ before $\text{com}(<v',x - 1>)$ and the latter before $\text{sub}(<v,x>)$, resulting in $ver(q) = x + 1$.

So suppose $r$ is a reconfiguration initiator. Proposition 5.1.2 shows that $r$ proposes $<v,x>$ if and only if $r$ detects $\text{GpView}^{x-1}$ as the most current $r$-defined system view. By definition then, for every $p$ in $\text{Acks}(r,\text{R-int}(x))$, either $ver(p) \geq x - 1$ or $\text{FAULTY}_r(p)$ holds.

Thus, $\text{FAULTY}_r(q)$ holds at the end of reconfiguration Phase I. In S-GMP, $q$ fails sympathetically immediately after receiving $\text{sub}(<v,x - 1>)$ from $r$, and then attempts to rejoin the group view as a new process.

Proposition 5.1.5 If $ver(r) = x$ then the most recent $r$-defined system view at the end of Reconfiguration Phase I is no less than $x - 1$ and no greater than $x$.

Proof $\text{GpView}^{x+1}$ (or any view greater than $x + 1$) cannot be the most recent $r$-defined system view as this would require $ver(r) \geq x + 1$ contradicting the hypothesis. So suppose $\text{GpView}^{x-2}$ is most recent. Let $q \in \text{Acks}(r,\text{R-int}(x))$ such that $ver(q) \leq x - 2$, and let $p$ be the process from which $r$ received $\text{com}(<v,x>)$. From Proposition 5.1.4, it is impossible for $q$ to have received and responded to
\text{sub}(<v, x>)$, so \text{FAULTY}_p(q) holds by the end of $p$'s submit phase. From Gossip, \text{FAULTY}_r(q) holds upon $r$'s receipt of \text{com}(<v, x>)}, meaning that $r$ cannot receive $q$'s response to \text{R-int}(x).

The temporal cohesion of outer members' local views guaranteed by Propositions 5.1.4 and 5.1.5 is fundamental to correctness. If non-faulty core members never receive submissions that omit an update, they cannot commit a view omitting an update. GMP-3 follows by showing successful submissions for the same group version are identical.

If $r$, with $\text{ver}(r) = x$, is successful in Reconfiguration Phase I, Proposition 5.1.1 tells us the largest version number observed among $r$'s respondents is $x + 1$. So suppose \text{Ahead}_r is non-null and let $p$ be a process from which some member of \text{Ahead}_r received \text{com}(<v_{x+1}, x + 1>). If the compressed update algorithm were used to install version $x + 1$, then every member of \text{Ahead}_r also responded to \text{sub}(<v_{x+2}, x + 2>). Moreover, $r$ can imagine that the processes in the complement, \overline{\text{Acks}(r, \text{R-int}(x))}, may also have done so making it possible that \text{Ahead}_r and \overline{\text{Acks}(r, \text{R-int}(x))} (and $v_{x+1}$ if $\text{GpView}^{r+1} = \text{GpView}^r \cup v_{x+1})$ together form the requisite majority subset of $\text{GpView}^{r+1}$ to commit view $x + 2$ (See Figure 4.6). Trouble can arise if \overline{\text{Acks}(r, \text{R-int}(x))} (and $v_{x+1}$ and $v_{x+2}$ if applicable) also constitutes a majority subset of $\text{GpView}^{r+2}$ since neither $r$ nor any process in \overline{\text{Acks}(r, \text{R-int}(x))} can know what value $p$ would propose for view $x + 3$. If removing core members is the only permitted operation, and $r$ is successful, it is impossible for another process to commit a version numbered higher than $x + 1$. When additions are per-
mitted, \( p \) may continue committing new group views and \( r \) may lag behind \( p \), but if both are successful for a given group version, both commit the same value.

**Proposition 5.1.6** Let \( r \) be a reconfiguration initiator with \( \text{ver}(r) = x \). Let \( \text{Ahead}_r \subseteq \text{Acks}(r, R-\text{int}(x)) \) report local version \( x + 1 \), \( \text{Acks}(r, R-\text{int}(x)) \) be a majority subset of \( \text{LocalView}_r^x \), and let \( p \) be a process from which some member of \( \text{Ahead}_r \) received \( \text{com}(<v_{x+1}, x+1>) \). Then, when the group view is only altered by removals, \( p \) cannot be successful for any view numbered higher than \( x + 1 \).

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

Observe that \( \text{GpView}^x = \text{LocalView}^x = \text{Acks}(r, R-\text{int}(x)) \cup \overline{\text{Acks}(r, R-\text{int}(x))} \), and that \( r \) will not receive \( \text{ack}(R-\text{int}(x)) \) responses from the members of \( v_{x+1} \); that is, \( v_{x+1} \subseteq \overline{\text{Acks}(r, R-\text{int}(x))} \). For \( p \) to commit \( \text{LocalView}_p^{x+2} \), \( \left( \text{Acks}(r, R-\text{int}(x)) - v_{x+1} \right) \) must be a majority subset of \( \text{LocalView}_p^{x+1} \). Then initiator \( p \) is successful for \( <v_{x+2}, x+2> \) if and only if

\[
\frac{|\text{Acks}(r, R-\text{int}(x))| - |v_{x+1}|}{|\text{LocalView}_p^{x+1}|} = \frac{|\overline{\text{Acks}(r, R-\text{int}(x))}| - |v_{x+1}|}{|\text{GpView}^x| - |v_{x+1}|} = \frac{|\text{Acks}(r, R-\text{int}(x))| - |v_{x+1}|}{|\text{Acks}(r, R-\text{int}(x))| + |\overline{\text{Acks}(r, R-\text{int}(x))}| - |v_{x+1}|} > \frac{1}{2} \iff \\
|\text{Acks}(r, R-\text{int}(x))| - |v_{x+1}| > |\text{Acks}(r, R-\text{int}(x))|\]


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

\textbf{Proposition 5.1.7} Let $r$ be a reconfiguration initiator with $\text{ver}(r) = x$. Let $\text{Ahead}_r$ be non-null and let $p$ be a process that sent $\text{com}(< v_{x+1}, x + 1 >)$ to some member of $\text{Ahead}_r$. Then if $r$ is successful for $< v_{x+1}, x + 1 >$:

1. If $p$ submits $< v_{x+2}, x + 2 >$ independent of the commit of $< v_{x+1}, x + 1 >$, then either $r$ or $p$, but not both, is successful for version $x + 2$.

2. If $p$ commits $< v_{x+1}, x + 1 >$ contingent upon $< v_{x+2}, x + 2 >$ being committed, then either $r$ or $p$, but not both, is successful for version $x + 3$.

\textbf{Proof}

1. Consider Figure 5.1 (top). The split-arrow message from $r$ to $v_{x+1}$ represents the two possibilities for the arrival of $r$’s commit message,

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

at $v_{x+1}$. $p$’s commit message, $\text{com}(< v_{x+1}, x + 1 >)$, to $v_{x+1}$ is a dashed line to represent the possibility that it may not be received. For presentation clarity, we elide $r$’s Phase II submit message.

GMP-3 would be violated if both $p$ and $r$ were able to commit their proposed values for version $x + 2$. 

(a) Suppose the members of $v_{x+1}$ receive $m$ from $r$ before they receive $\text{com}(<v_{x+1}, x+1>)$ 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 $\text{M-sub}(<v_{x+2}, x+2>)$. We say $r$ owns $v_{x+1}$.

Then $p$ is successful for version $x+2$ if and only if

$$|\text{Acks}(r, \text{R-int}(x))| > |\text{Acks}(r, \text{R-int}(x))| + |v_{x+1}|$$

and $r$ is successful for version $x+2$ if and only if

$$|\text{Acks}(r, \text{R-int}(x))| + |v_{x+1}| > |\text{Acks}(r, \text{R-int}(x))|$$

Both conditions cannot hold.

(b) The analysis is the same as the previous case; $r$ owns $v_{x+1}$ once the members of that set receive $m$.

(c) In the last case, $p$ owns $v_{x+1}$ if its $\text{M-sub}(<v_{x+2}, x+2>)$ message arrives at $v_{x+1}$ before $r$'s $\text{R-com}(\cdot)$ message, and so gossips $p$'s belief in $r$'s faultiness. Then $p$ is successful for version $x+2$ if and only if

$$|\text{Acks}(r, \text{R-int}(x))| + |v_{x+1}| > |\text{Acks}(r, \text{R-int}(x))|$$

and $r$ is successful for version $x+2$ if and only if

$$|\text{Acks}(r, \text{R-int}(x))| > |\text{Acks}(r, \text{R-int}(x))| + |v_{x+1}|$$

Again, both conditions cannot hold.
2. There are six cases to consider, but as the analyses are quite similar, we depict and discuss only four (Figures 5.2 and 5.3).

(a) Suppose \( r \) owns \( v_{x+1} \). This happens if the members of \( v_{x+1} \) receive \( r \)'s commit message for version \( x + 1 \),

\[
m^{x+1}_r = \text{R-com}(< v_{x+1}, x + 1 >) : \text{M-sub}( < v_{x+2}, x + 2 >),
\]

before either \( p \)'s commit message for version \( x + 1 \),

\[
m^{x+1}_p = \text{com}(< v_{x+1}, x + 1 >) : \text{sub}( < v_{x+2}, x + 2 >),
\]

or \( p \)'s commit message for version \( x + 2 \),

\[
m^{x+2}_p = \text{com}(< v_{x+2}, x + 2 >) : \text{M-sub}( < \neg \text{Acks}(r, \text{R-int}(x)), x + 3 >)
\]

(Hence, the split-arrow). Further suppose \( r \) owns \( v_{x+2} \), as happens when the members of \( v_{x+2} \) receive

\[
m^{x+2}_r = \text{R-com}( < v_{x+2}, x + 2 >) : \text{M-sub}( < \neg \text{Acks}(r, \text{R-int}(x)), x + 3 >)
\]

from \( r \) before receiving \( m^{x+2}_p \) from \( p \). Then \( r \) is successful for \( x + 3 \) if and only if

\[
| \text{Acks}(r, \text{R-int}(x)) | + | v_{x+1} | + | v_{x+2} | > | \overline{\text{Acks}(r, \text{R-int}(x))} |
\]

and \( p \) is successful for \( x + 3 \) if and only if

\[
| \overline{\text{Acks}(r, \text{R-int}(x))} | > | \text{Acks}(r, \text{R-int}(x)) | + | v_{x+1} | + | v_{x+2} |
\]

Both conditions cannot hold.
(b) Suppose $p$ owns $v_{x+1}$ and $v_{x+2}$. Then $p$ is successful for $x+3$ if and only if

$$\left| \text{Acks}(r, \text{R-int}(x)) \right| + \left| v_{x+1} \right| + \left| v_{x+2} \right| > \left| \text{Acks}(r, \text{R-int}(x)) \right|$$

and $r$ is successful for $x+3$ if and only if

$$\left| \text{Acks}(r, \text{R-int}(x)) \right| > \left| \overline{\text{Acks}(r, \text{R-int}(x))} \right| + \left| v_{x+1} \right| + \left| v_{x+2} \right|.$$ 

(c) In this situation (Figure 5.3), $p$ owns $v_{x+1}$ and $r$ owns $v_{x+2}$. Then $p$ is successful for $x+3$ if and only if

$$\left| \text{Acks}(r, \text{R-int}(x)) \right| + \left| v_{x+1} \right| > \left| \text{Acks}(r, \text{R-int}(x)) \right| + \left| v_{x+2} \right|$$

and $r$ is successful for $x+3$ if and only if

$$\left| \text{Acks}(r, \text{R-int}(x)) \right| + \left| v_{x+2} \right| > \left| \overline{\text{Acks}(r, \text{R-int}(x))} \right| + \left| v_{x+1} \right|.$$ 

(d) In the last scenario $p$ owns $v_{x+2}$ and $r$ owns $v_{x+1}$. Then $p$ is successful for $x+3$ if and only if

$$\left| \overline{\text{Acks}(r, \text{R-int}(x))} \right| + \left| v_{x+2} \right| > \left| \text{Acks}(r, \text{R-int}(x)) \right| + \left| v_{x+1} \right|$$

and $r$ is successful for $x+3$ if and only if

$$\left| \text{Acks}(r, \text{R-int}(x)) \right| + \left| v_{x+1} \right| > \left| \overline{\text{Acks}(r, \text{R-int}(x))} \right| + \left| v_{x+2} \right|.$$
Cases a, and b: r owns \( v_{x+1} \).

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

Figure 5.1: Case Analysis for Proposition 5.1.7, Part 1.
Case a: $r$ owns $v_{x+1}$ and $v_{x+2}$.

Case b: $p$ owns $v_{x+1}$ and $v_{x+2}$.

Figure 5.2: Case Analysis for Proposition 5.1.7, Part 2.
Case c: $p$ owns $v_{x+1}$, $r$ owns $v_{x+2}$.

Case d: $r$ owns $v_{x+1}$, $p$ owns $v_{x+2}$.

Figure 5.3: Continued Case Analysis for Proposition 5.1.7, Part 2.
Proposition 5.1.7 is a keystone to correctness because it addresses exactly the situations in which it might appear possible for s-GMP to violate GMP-2 and GMP-3: when two initiators are able to propose, and therefore might be able to commit, different values for the same group version. Propositions 5.1.6 and 5.1.7 prove that even when the local views of processes installing the same group view differ (i.e. the initiators do not vie for majorities from among the same set of core members) s-GMP is safe.

It remains to consider how r decides which of the various proposals it learns about could have been invisibly committed. In s-GMP this necessity arises during reconfiguration in determining v2 when either \( \text{Ahead}_r \neq \emptyset \) or \( \text{Ahead}_r = \emptyset \) and \( \text{Behind}_r \neq \emptyset \), and in determining v1 when \( \text{Ahead}_r = \text{Behind}_r = \emptyset \).

To illustrate the difficulty, suppose \( \text{GpView}^{r-1} \) is the most recent \( r \)-defined group view. Let \( \text{Submissions}_r(x) \) be the set of proposed next updates for version \( x \) that \( r \) learns about in response to its \( \text{R-int()} \) message:

\[
\text{Submissions}_r(x) = \{ v_z \mid \exists p \in \text{Acks}(r, \text{R-int()}). \text{NextUpdate}_p = [v_x, x, z] \}
\]

for some initiator \( z \).

Intuitively, if \( \text{GpView}^{r-1} \) were committed with an accompanying proposal for \( \text{GpView}^r \) (i.e. the compressed algorithm were used), then \( \text{Submissions}_r(x) \) would be that proposal and \( |\text{Submissions}_r(x)| = 1 \). However, it is also possible that, while there may have been no plans for a future update when \( \text{mgr} \) multicast \( \text{M-com}(v, x - 1) \), \( \text{mgr} \) did begin an update algorithm to form \( \text{GpView}^r \) at some later time. If, during that same interval, an outer core member, \( p \), began reconfig-
uration it is possible for $p$ not to receive any Phase I responses indicating $mgr$'s intention for $GpView^x$. In this case, $p$ would propose $mgr$'s removal for version $x$. A subsequent reconfigurer could then get Phase I responses indicating both of these proposals. We first describe the composition of $\text{Submissions}_{r}(x)$, showing that every reconfigurer proposing version $x$ either propagates $mgr$'s proposal for version $x$ or proposes $mgr$'s removal (See Figure 4.1 in Chapter 4).

**Proposition 5.1.8** For all versions $x$,

$$|\text{Submissions}_{r}(x)| \leq 2.$$ 

**Proof** Inspecting procedure $\text{DetermineProposal}$, 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 $\text{Submissions}_{r}(x) = \emptyset$.

The set $\text{Submissions}_{r}(x)$ is bivalent if it contains two distinct values. Corollary 5.1.1 follows by examining $\text{DetermineProposal}$.

**Corollary 5.1.1** Let $r$ and $r'$ be reconfigurers proposing version $x$. Then if both are bivalent, they are identical:

$$\left( |\text{Submissions}_{r}(x)| = |\text{Submissions}_{r'}(x)| = 2 \right) \implies \text{Submissions}_{r}(x) = \text{Submissions}_{r'}(x).$$
That is all reconfigurers either propagate $mgx$'s unique submission for view $x$ or propose $mgx$'s removal. We now show that only one of the two proposals for a given version could possibly have been committed (invisibly or otherwise), and that all reconfigurers can distinguish which of the two it was. This proposition, too, is important in 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 5.1.9** Let $r$ be a reconfiguration initiator. If $Acks(r, R-int())$ is a majority subset of $LocalView_r$, and $Submissions_r(x)$ 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 $< v, x >$ and $< v', x >$ be the two elements of $Submissions_r(x)$. 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 $< v', x >$. Then $r$ reasons as follows: In order for either value to have been committed its initiator must have garnered majority approval from its own local view for the submitted value. That is,

$$Acks(p, sub(< v, x >)) \in \text{Maj}(LocalView_p) \quad \text{and/or} \quad Acks(p', sub(< v', x >)) \in \text{Maj}(LocalView_{p'}).$$

Since both $p$ and $p'$ make version $x$ submissions, both must have local versions either $x - 1$ or $x$ (Proposition 5.1.3). We consider these by cases:
1. If \( \text{ver}(p) = \text{ver}(p') \), then both processes vie for majority approval from among the same set of processes, and the Phase I response sets of both must intersect:

\[
\text{Acks}(p, R\text{-}\text{int}(\text{ver}(r))) \cap \text{Acks}(p', R\text{-}\text{int}(\text{ver}(p'))) \neq \emptyset.
\]

Suppose \( p \) has higher rank than \( p' \). Then \( p \) can only get a majority response to its Phase I and Phase II messages if it does so before \( p' \).\(^1\) If this were the case, then \( p' \) would have learned of \( \text{sub}(< v, x >) \) and have propagated it (c.f. \text{DetermineProposal}). However, \( p' \) submitted \( < v', x > \), and \( v' \neq v \), so \( p \) must not have been successful for \( < v, x > \).

2. If \( \text{ver}(p) \neq \text{ver}(p') \), then one of \( p \) or \( p' \) has version \( x \) and the other has version \( x - 1 \) (Proposition 5.1.3). Suppose \( \text{ver}(p) = x \) and let \( r_0 \) be the process from which \( p \) received \( \text{com}(< v, x >) \).

Because it, too, proposed version \( x \), \( r_0 \)'s local version must also be either \( x - 1 \) or \( x \). Suppose \( \text{ver}(r_0) = x - 1 \), and consider the conditions under which \( p' \) proposes \( < v', x > \):

(a) If \( \text{Submissions}_{p'}(x) \) is the singleton \( \{ v' \} \), or if \( \text{Submissions}_{p'}(x) \) is null and \( v' \) is the singleton \( \{ \text{mgr} \} \). In either case, all members of

\[
\text{Acks}(p', R\text{-}\text{int}(x - 1)) \cap \text{Acks}(r_1, \text{sub}(< v, x >))
\]

\(^1\)It is not difficult to see that when two processes compete for majorities from among the same process pool, one must succeed in getting the majority before the other. Moreover, Disconnect, Gossip, and \text{INITIATE()} mean that the process with higher rank can only get a majority response if it does so before the lower-ranked process.
must have received $p'$'s $R$-$\text{int}(x - 1)$ before $r_0$'s $\text{sub}(v, x)$; if not, they would have forwarded $<v, x>$ in their responses to $p'$ making $\text{Submissions}_{p'}(x)$ bivalent. Moreover, because they received $\text{sub}(v, x)$ from $p'$, $\text{rank}(r_0) < \text{rank}(p')$. Since $\text{Acks}(r_0, \text{sub}(v, x))$ is a majority subset of $\text{GpView}^{x-1}$, a majority of $\text{GpView}^{x-1}$ believe $p'$ faulty upon receipt of $\text{sub}(v, x)$; $p'$ could not have obtained the requisite majority for its submission $R$-$\text{sub}(v', x)$.

(b) If $\text{Submissions}_{p'}(x)$ is bivalent then $\text{Submissions}_{p'}(x) = \{v', v\}$. Suppose $p'$ is the ‘first’ reconfigurer for which $|\text{Submissions}_{p'}(x)| = 2$. Among the responses to $R$-$\text{int}(x - 1)$ indicating $<v, x>$, let $r_1$ be the member of least rank to propose $<v, x>$. Similarly, let $r_2$ be the member of least rank that proposed $<v', x>$. By assumption, both $\text{Submissions}_{r_1}(x)$ and $\text{Submissions}_{r_2}(x)$ are either null or a singleton.

Without loss of generality, suppose $<v, x>$ is the proposal to remove $mgr$ (one of the two submissions must be). Then $\text{Submissions}_{r_1}(x)$ is either empty or the singleton, $\{<mgr, x>\}$, and $r_2$ must be the singleton $\{<v', x>\}$. We now show $p'$ can deduce $r_2$ cannot have been successful for $<v', x>$ by considering the possible roles $r_2$ might have played.

i. If $r_2$ were $mgr$ we claim it could not have received a majority response from among $\text{GpView}^{x-1}$; if it had, then $r_1$ would have learned of $<v', x>$ during its interrogation phase and thence propagated it.

ii. So consider $r_2 \neq mgr$. Since both $r_1$ and $r_2$ were able to propose
updates, their Phase I respondents sets must intersect, and since the processes in that intersection received $R\text{-int}(x - 1)$ from $r_1$, it must be the case that $\text{rank}(r_1) < \text{rank}(r_2)$.

Furthermore, had $r_2$ been able to commit $< v', x >$ its Phase II respondents, $\text{Acks}(r_2, R\text{-sub}(< v', x >))$, would have had to have been a majority subset of $\text{GpView}^{x-1}$. In this way, if $r_2$ had been successful for $< v', x >$.

$$\left( \text{Acks}(r_2, R\text{-sub}(< v', x >)) \cap \text{Acks}(r_1, R\text{-int}(x - 1)) \right) \neq \emptyset.$$

Since $\text{Submissions}_{r_1}(x)$ does not contain $< v', x >$, the processes in the intersection must have received $r_1$’s $R\text{-int}(x - 1)$ before $r_2$’s $R\text{-sub}(< v', x >)$. But now, $\text{rank}(r_1) < \text{rank}(r_2)$ means that having received $r_1$’s $R\text{-int}(x - 1)$, no process will receive further messages from $r_2$ so $r_2$’s $R\text{-sub}(< v', x >)$ could not have reached the required majority.

Thus, $p'$ deduces $r_2$ cannot have been successful for $< v', x >$, and $p'$ should decide to propagate $< v, x >$. Examining procedure $\text{GetStableProposal}$, we see that this is indeed the case: $< v, x >$ is submitted by the process of least rank, $r_1$, among those reported in $\text{Submissions}_{p'}(x)$. We have arrived at a contradiction: $\text{DetermineProposal}$ and $\text{GetStableProposal}$ would not have resulted in $p'$ submitting $< v', x >$, as hypothesized.

We have just proven the claim for the case when $p'$ is the first recon-
figurer whose Phase I response set is bivalent. If \( p' \)'s proposal reaches a majority subset then this value will be chosen by subsequent initiators (with bivalent response sets) in \( \text{GetStableProposal} \) since now \( p' \) is the submitter with least rank. If \( p' \)'s proposal does not reach a majority subset, subsequent initiators will nonetheless reason as \( p' \) did and so also choose to propagate the correct value.

Finally, recall that \( r_0 \) is the process from which \( p \) received \( \text{com}(<v,x>) \). We have just proven the proposition for the case when \( \text{ver}(r_0) = x - 1 \). The last case to consider is \( \text{ver}(r_0) = x \). But in this case we can apply the same argument to the process, say \( \hat{r} \) from which \( r_0 \) received \( \text{com}(<v,x>) \); this reduction and analysis can continue for only a finite number of steps before we exhaust the members of \( \text{GpView}^{x-1} \).

Proposition 5.1.9 proves that \( \text{GetStableProposal} \) 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 \( \text{DetermineProposal} \) is safe.

Stably-defined proposals (those submitted by successful initiators) are exactly the proposals that reconfigurers must view as possibly committed invisibly.

**Corollary 5.1.2** If \( \text{GpView}^{x-1} \) is defined, there is at most one stably-defined proposal for group version \( x \).
Proof Proposition 5.1.9 proves that any reconfigurer reaching its proposal stage knows exactly which of the two proposals for version \( x \) is not stably-defined. Procedures \textit{DetermineProposal} and \textit{GetStableProposal} propagate the other one. If this initiator reaches its commit stage, its proposal is stably-defined and identical to the other stably defined proposals for version \( x \).

\[ \text{Theorem 5.1.1 (Identical Local Views)} \] If \( \text{GpView}^{x-1} \) is defined, then all members that survive to define local version \( x \) have identical local \( x^{th} \) views:

\[
\bigwedge_{0 \leq x} \bigwedge_{p, q \in \text{GpView}^{x-1}} \left( (\diamond \text{ver}(p) = x) \land (\diamond \text{ver}(q) = x) \right) \Rightarrow \\
\left( \text{LocalView}_{p}^{x} = \text{LocalView}_{q}^{x} \right).
\]

Proof The result follows from Corollary 5.1.2; 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 5.1.1 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 \( \text{LocalView}_{p}^{x} \) and \( \text{LocalView}_{q}^{x} \) to exist contemporaneously. We need to do a bit more work to prove GMP-3.

Remarks

\( S\text{-GMP} \) ensures that the state to which the system finally reconfigures represents the \textit{cumulative} system progress. It accounts for any previous updates (and reconfigura-
tions) that may have been only partially successful, and makes them stable. With respect to an interrupted commit the group view is undefined until some initiator succeeds in multicasting its commit message.

To see that the new $mgr$ is unique consider a process, $p$, that has received an interrogate message from $r$. It believes faulty and disconnects from every process ranked higher than $r$. Thus, $p$ immediately begins to believe that $r$ is the highest ranking non-faulty process. In particular, $p$ will not receive submit, commit, or interrogate messages from processes ranked higher than $r$.

5.2 S-GMP Correctness Proofs

Theorem 5.2.1 S-GMP satisfies GMP-0.

Proof Let $p$ be a process that queries the name server and is told no MRM exists. Then $LocalView_p(c_0) = GpView_{[p]}(c_0) = \{p\}$. Uniqueness of $GpView_{[p]}(c_0)$ depends on race conditions and assumptions on the name server. Until the MRM reaches its critical mass and can be used to create a fault-tolerant name service, its reliance on the name service makes its uniqueness vulnerable at start-up.

Theorem 5.2.2 S-GMP satisfies GMP-1.

Proof A process, $p$, executes $remove_p(q)$ only upon receipt of one of the following:

1. $M-com(-q):M-sub(-q')$ from $mgr$, in which case S-GMP gives

$$recv_p(mgr,M-sub(-q)) \rightarrow faulty_p(q) \rightarrow$$
\[ \text{recv}_p(mgr, \text{M-com}(-q): \text{M-sub}(-q')) \rightarrow \text{remove}_p(q). \] (5.2)

Equation 5.2 holds irrespective of whether the submit message \text{M-sub}(-q) were the tail end of a contingent update or submitted independently of any accompanying commit message.

2. \text{R-com}(-q):\text{M-sub}(q') from some reconfiguration initiator, r. In this case

\[ \text{recv}_p(r, \text{R-sub}(-q)) \rightarrow \text{faulty}_p(q) \rightarrow \text{recv}_p(r, \text{R-com}(-q)) \rightarrow \text{remove}_p(q). \]

Analogous statements apply to the events \text{operating}_p(q) and \text{add}_p(q).

The next two theorems will make use of the following consistent cuts. Let \( r_x \) be the process responsible for completing the MRM-wide commit of \(< v_x, x >\). Theorems 5.2.3 and 5.2.4 will show that \( c_x \) is the first cut (in the sense of the partial ordering, \( \leq \), on cuts) along which \( \text{GpView}^x \) exists. \( c_x[p] \) defines the \( p \)-component of \( c_x \) for \( p \in \text{GpView}^x \).

\[
c_x[p] \overset{\text{def}}{=} \left\{ \begin{array}{l}
\text{recv}_p(r_x, \text{com}(< v_x, x >)) \quad \text{DoCommit}(< v_x, x >) \rightarrow \\
\text{recv}_p(r_x, \text{com}(< v_x, x >)) \\
\text{DoCommit}(< v_x, x >) \quad \text{recv}_p(r_x, \text{com}(< v_x, x >)) \rightarrow \\
\text{DoCommit}(< v_x, x >) \\
\text{quit}_p \quad \text{otherwise}
\end{array} \right.
\]

**Theorem 5.2.3** \( s\text{-GMP} \) satisfies GMP-2.
Proof Recall GMP-2:

\[ \bigwedge_{c} \bigvee_{S \subseteq \text{Proc}} \left( \text{GpView}_{S}(c) = S \right) \implies \bigwedge_{\emptyset \neq S' \neq S} \text{UNDEF}'(\text{GpView}_{S'}(c)), \]

and define the \( x \)th interval of a run as follows:

\[ [c_x, c_{x+1}) = \{ c \mid c_x \leq c < c_{x+1} \}. \]

We prove the theorem by induction over intervals.

**Base Case** Suppose exactly one process, \( p \), is installed as the initial group view, \( \text{GpView}^0 \). Let \( S_0 = \{ p \} \). Then until \( p \) changes its local view (that is, along every \( c \in [c_0, c_1) \)),

\[ \text{GpView}_{S_0}(c) = \{ p \} = S_0. \]

By assumption, every other process, \( q \), that queries the name server during this interval is told the MRM exists, so \( \text{LocalView}_q(c) \) is undefined until \( q \) is allowed to join and this is no sooner than \( c_1 \).

Caveat: The Base Case holds only when the MRM encounters no race conditions at start-up.

**Inductive Hypothesis** The statement of the theorem holds during the interval \([c_{x-1}, c_x)\), with \( S_{x-1} \) the unique set satisfying

\[ \text{GpView}_{S_{x-1}}(c_{x-1}) = S_{x-1} = \text{GpView}^{x-1} \]

and no other set satisfying the equality along any cut in the interval.
**Inductive Step** From Corollary 5.1.2, if $\text{GpView}^z_{x-1}$ is uniquely defined, then there is a unique, stably-defined proposal, $<v_x, x>$, for $\text{GpView}^z$. Define $S_x \subseteq \text{Proc}$:

$$S_x = S_{x-1} \pm v_x.$$ 

Then, by definition

$$q \in \text{GpView}_{S_x}(c_x) \iff \bigwedge_{p \in (S_x \cap \text{Up}(c_x))} q \in \text{LocalView}_p(c_x).$$

Consider the possibilities for $p$. If $p \in \text{GpView}^{z-1}$, then

$$\text{LocalView}_{p'}(c_x) = \text{LocalView}_{p'}(c_{x-1}) \pm v_x.$$ 

Applying the Inductive Hypothesis, we get

$$\text{LocalView}_{p'}(c_{x-1}) \pm v_x = \text{GpView}^{z-1} \pm v_x = S_{x-1} \pm v_x = S_x.$$ 

On the other hand, if $p$ is a process that was added to $\text{GpView}^{z-1}$ to form $\text{GpView}^z$ (that is $p \notin \text{GpView}^{z-1}$) then $\text{LocalView}_{p'}(c_x)$ is determined by (and equals the local view of) the initiator that sent $p$ the \text{Join} and \text{State-Xfer} messages. Let $r$ be this initiator. Now since $r \in \text{GpView}^{z-1}$, the Inductive Hypothesis gives

$$\text{LocalView}_{p'}(c_x) = \text{LocalView}_r(c_x) = S_{x-1} \pm v_x = S_x.$$ 

That is, $q \in \text{GpView}_{S_x}(c_x) \iff q \in S_x$.

To see that $S_x$ uniquely satisfies the equality, note both $S_{x-1}$ and $<v_x, x>$ are unique. Further, if $<v_x, x>$ is the removal of processes from $\text{GpView}^{z-1}$,
those processes’ local views, if they exist at all, are equal to $GpView^{x-1}$. Since $GpView^{x-1}$ contains all the members of $S_x = GpView^x$, $GpView_{v_x}(c)$ is undefined for $c \in [c_x, c_{x+1})$ – the local views of the functional members of $GpView^{x-1} - v_x$ differ from the local views of the members of $v_x$ along $c_x$.

In addition any process that queries the name server during this interval cannot define a local view until it is allowed to join the MRM core.

Finally, from the definition of the partial order $\leq$ on cuts, $\bigcup_{0 \leq i} [c_i, c_{i+1})$ covers the entire asynchronous run.

\[ \textbf{Theorem 5.2.4} \quad S\text{-GMP satisfies GMP-3.} \]

\textbf{Proof} Recall GMP-3:

\[
\bigwedge_{0 \leq x} \bigwedge_{p} \bigtriangleup \bigwedge_{q} \left( \neg \text{IN-LOCAL}_q^{x}(q) \Rightarrow \text{UNDEF'D}(\text{LocalView}_q^{x}) \right) \land \\
\left( \text{IN-LOCAL}_q^{x}(q) \Rightarrow \text{DOWN}_q \lor \left( \text{LocalView}_q = \text{LocalView}_p^{x} \right) \right).
\]

Let $c_x$ be as defined above and let $\text{LocalView}_p^{x}(c_x)$ exist and be $\text{LocalView}_p^{x}$. If $q \in \text{LocalView}_p^{x-1} - \text{LocalView}_p^{x}$ then $q$ will not commit any version $\text{LocalView}_q^{x+i}$, for $0 \leq i$; once $q$ receives $\text{com}(<v_x, x>)$ it learns $v_x$ is its own removal from $GpView^{x-1}$ and crashes. If $q$ is in neither $\text{LocalView}_p^{x-1}$ nor $\text{LocalView}_p^{x}$, then $\text{LocalView}_q^{x}$ is undefined along $c_x$. 

So suppose \( q \in \text{LocalView}^r_p \). If \( q \) crashed on or before \( c_x \) we are done. If \( q \) has not crashed the result follows from Theorem 5.1.1 and the definition of \( c_x \).

It remains to prove \( c_x \) consistent. By way of contradiction, suppose it is not, and let \( e_p \) and \( e_{p'} \) be events violating consistency:

\[
e_p \rightarrow_1 e_{p'}, \ e_{p'} \in c_x, \text{ and } e_p \notin c_x.
\]

\( p \) and \( p' \) must be different different processes, making \( e_p \) a send event from \( p \) to \( p' \) and \( e_{p'} \) the corresponding receive event. Consider Figures 5.4 and 5.5.

\textbf{a: } \( p = r_x, p' \neq r_x \). The FIFO assumption precludes this (Figure 5.4-a).

\textbf{b: } \( p \neq r_x, p' = r_x \). If \( c_x \) is inconsistent, then following cycle ensues, violating causality (Figure 5.4-b):

\[
\text{send}_{p'}(p, R-\text{com}(< v, x >)) \rightarrow \text{recv}_p(r, R-\text{com}(< v, x >)) \rightarrow c_x[p] \rightarrow
\]

\[
e_p = \text{send}_p(p', m) \rightarrow e_{p'} = \text{recv}_{p'}(p, m) \rightarrow \text{send}_{p'}(p, R-\text{com}(< v, x >)).
\]

\textbf{c: } \( p \neq r_x, p' \neq r_x \). If \( p \) is a subsequent initiator, it must have lower rank than \( r_x \), making it impossible for \( p' \) to receive \( r_x \)'s commit message (Figure 5.5-c1 and c2).

\textbf{d: } \( p \neq r_x, p' \neq r_x \). Finally, if \( p' \) is a subsequent initiator, then \( p \) is responding to a message from it. In no case, however, will \( p' \) receive the commit message from \( r_x \); if \( p' \) has lower rank than \( r_x \), \( \text{FAULTY}_{p'}(r_x) \) holds, and if \( p' \) has higher rank than \( r_x \), \( p' \) will fail upon receipt of \( r_x \)'s interrogate message, and not receive its commit message (Figure 5.5-d).
Figure 5.4: Proving $c_x$ Consistent - Theorem 5.2.4
**Theorem 5.2.5** S-GMP satisfies GMP-4.

**Proof** An outer process’s local failure and recovery beliefs are propagated by Gossip in every message it sends. Thus, an update initiator learns of, adopts, then acts on outer processes’ beliefs.

\[ \square \]

### 5.3 Message Complexity

The sequence and timing of failures affect the message complexity of S-GMP. We quantify the worst and best case complexity for our protocol to install a new group view, as well as the gain achievable when we can take advantage of compressing phases.

Define \( n_x \overset{\text{def}}{=} | \text{GpView}^x | \), and \( \tau_x \) to be the number of tolerable failures in \( \text{GpView}^x \):

\[
\tau_x \overset{\text{def}}{=} \left( \left\lfloor \frac{n_x}{2} \right\rfloor + 1 \right).
\]

Then the worst case scenarios to install the \((x + 1)^{st}\) group view occurs when there are \( \tau_x \) successive failed (or aborted) reconfigurations. This results in

\[
\sum_{y=1}^{\tau_x} 5 \left( (n_x - 1) - (y - 1) \right) = 5n_x \tau_x - \frac{5}{2} (\tau_x)^2 - \frac{5}{2} \tau_x = O\left( | \text{GpView}^x |^2 \right)
\]

messages. Fortunately, this specific composition, ordering, and timing of failures occurs with extremely low probability.
There are three so-called best case scenario to install a view: \( mgr \) succeeds using the standard two-phase S-GMP; \( mgr \) succeeds using compressed S-GMP; and one successful reconfigurer. The first case requires at most \( 3(n_x - 1) \) messages; the second, at most \( 2(n_x - 1) \); and the third, at most \( 5(n_x - 1) \).

Finally, if we can take advantage of a sequence of compressed S-GMP updates, we save substantially in message complexity. For example, \( n_x - 1 \) successive singleton removals, none of which are \( mgr \), requires

\[
(n_x - 1) + 2 \sum_{y=2}^{n_x-1} (n_x - y) = (n_x - 1) + 2n_x(n_x - 2) - 2\left(\sum_{y=2}^{n_x-1} y\right) =
\]

\[
n_x^2 - 2n_x - 1 = (n_x - 1)^2
\]

messages, averaging to \( n_x - 1 \) messages per update. A standard two-phase algorithm would require an additional \( \frac{n_x - y}{2} - 1 \) messages for the \( y^{th} \) removal, on the average. In each case, the number of actual failures may reduce the number messages.
$r_x$

$p$

$p'$

$c_1$: $p$ subsequent initiator

$\text{rank}(p) < \text{rank}(r_x)$

$r_x$

$p$

$p'$

$c_2$: $p$ subsequent initiator

$\text{rank}(r_x) < \text{rank}(p)$

$r_x$

$p$

$p'$

$\text{rank}(r_x) = \text{rank}(p)$

$
\text{rank}(r_x) = \text{rank}(p)
$

$d$: $p'$ subsequent initiator

Causality violation

Figure 5.5: Proving $c_x$ Consistent – Theorem 5.2.4 continued.
Chapter 6

Protocol Optimality

In this chapter, we rephrase the Strong GMP membership problem as a knowledge-theoretic, commit-style problem, viewing any solution to GMP as one of acquiring and propagating knowledge [HM90]. This approach proved superior to the previous, behavioral specification with respect to fine-tuning the $s$-GMP protocol. We show here how the epistemic formulation makes clear that any solution to Strong GMP requires majority corroboration, and where optimizations to the $s$-GMP protocol are and are not possible. We further show the knowledge-based formulation facilitates establishing a lower bound on the number of messages needed to solve Strong GMP, construct a message-minimal solution, and finally quantify the tradeoffs between the given, optimized solution and the message-minimal solution.

Others have used a knowledge-based approach to analyze a variety of problems in distributed computing ([Hal87], [Had91], [MF90], [Maz90], [HF85]). The work presented here differs both in the problem considered and in demonstrating
the practical utility of an epistemic formulation. We give concrete evidence that such a formulation greatly simplifies building robust, fast solutions to commit-style problems.

**Remarks**

The indistinguishability of failures from communication delays seems to warrant using the doxastic modality [Tut89] to refer to local failure beliefs. However, the *Gossip* and *Isolation* properties result in a much stronger local interpretation than belief. In particular, once a process suspects another is faulty, it behaves as if it knows that process is faulty. Whereas the standard doxastic interpretation would give equal weight to belief in faultiness and belief in non-faultiness, our model favors one before the event \( fault_y_p(q) \) and the other after. By closing its incoming channel from a suspected-faulty process, a process behaves as if the system were synchronous; as if it knew that a suspected process could not send further messages.

Note that the formulae \( \text{FAULT}_y_p(q) \) and \( \text{DOWN}_q \) reflect this ambiguity. The former is *local* to the suspecting process while the latter is local to no process. Formula \( \phi \) is local to \( p \) if \( p \) always knows whether it is true [CM86]: \( K_p\phi \lor K_p\neg\phi \) holds along all consistent cuts. In this way, the standard knowledge operator models exactly the behavior process \( p \) exhibits upon executing \( fault_y_p(q) \). Throughout this chapter, the statement “\( p \) believes (or knows) \( q \) is faulty” should be taken as an artifact of our model’s behavioral requirements, not literally.

In contrast Chandra and Toueg impart a pure doxastic interpretation to local failure beliefs [CT91]. In their work, despite belief in a process’s faultiness, all of
its messages must be delivered, as must all messages to it. The difference could also be termed one of optimism versus pessimism.

Finally, we note that S-GMP is a full-information protocol; all relevant state information is sent with each protocol message. In Strong GMP, relevant information are local beliefs about failures, and during reconfiguration, a process’s local view and the value of a pending commit (if it exists). Outer processes can infer an initiator’s respondents from its failure beliefs.

6.1 Reformulating Strong GMP

Strong GMP, as a commit problem, requires functional core members in a given group view to vote on a proposed update to their local views, and commit the update when particular conditions are met.

Let \( V \) be a set of values, one of which the processes in the \((x-1)^{st}\) group view must commit to install the \(x^{th}\) group view: \( V \subseteq \{-, +\} \times 2^{\text{Proc}} \). Before committing a given update, a process may vote for more than one update value. As in Atomic Commit, in no execution are votes for any version pre-determined. The notation \(<v, x>\) refers to a particular value, \(v\), and version of the group view for which \(v\) is submitted and/or committed. The same value may be submitted for different versions.

The following are formulae, events, and notation used to rephrase Strong GMP as a commit problem. We say a formula \(\phi\) is local to a process \(p\) if \(p\) always knows whether \(\phi\) is true; that is:

\[ K_p\phi \lor K_p\neg\phi \]
always holds.

- **VOTE\(_p\)(<\(v, x>\))** holds along consistent cut \(c\) if \(votep(<v, x>)\) is the most-recent voting event \(p\), with local view \(LocalView^p_{x-1}\), executed.

- **COMMIT\(_p\)(<\(v, x>\))** holds if \(p\) executed the commit event \(commitp(<v, x>)\) to form its \(x^{th}\) local view.

The formulae **FAULTY\(_p\)(q)**, **VOTE\(_p\)(<\(v, x>\))**, and **COMMIT\(_p\)(<\(v, x>\))** are local to \(p\).

- **Maj(S)** is the set of all majority subsets of \(S\), and **Sizeof-Maj(S)** (or sometimes \(\mu(S)\)) is the size of the smallest majority subset of \(S\):

\[
\text{Sizeof-Maj}(S) = \mu(S) = \left\lfloor \frac{|S|}{2} \right\rfloor + 1.
\]

- The epistemic operator for distributed knowledge of a formula, \(\psi\), among a set of processes, \(S\), captures the notion that if the processes of \(S\) pooled their local knowledge together, they would know \(\psi\). Formally, let \(\Phi_p(c)\) be the set of all the formulae \(p\) knows along \(c\):

\[
\Phi_p(c) = \{ \phi \mid c \models K_p \phi \}
\]

Then

\[
c \models D_S\psi \iff c \models \left( \bigwedge_{p \in S} \Phi_p(c) \right) \Rightarrow \psi.
\]

- **STABLEVOTE\(_S\)(<\(v, x>\))** holds when \(S\) is a (non-null) subset of \(GpView^x-1\), each process of which has most-recently voted for \(<v, x>\) and will not vote
for a different \( <v', x> \). Moreover, every process not in \( S \) is (distributedly) known faulty by the group \( S \):

\[
\text{\textbf{STABLEVOTES}}_S(<v, x>) \overset{\text{def}}{=} \bigwedge_{p \in S} (\text{VOTE}_p(<v, x>) \land \bigwedge_{v' \neq v} \neg \text{VOTE}_p(<v', x>)) \land \bigwedge_{q \not\in S} D_S \text{FAULTY}(q)
\]

Observe that the set \( S \) must have at least one functioning process in order for \( \text{STABLEVOTES}_S(<v, x>) \) to hold. We omit the subscript when we are not concerned with the particular set according to which \( <v, x> \) is stable.

Commit problems are generally characterized by two safety conditions: \textit{Validity} describes the voting conditions (e.g. number of processes voting) that must exist when a value is committed, and \textit{Uniqueness} describes the commit conditions (e.g. whether another value can have been committed) that must exist. For Strong GMP, Validity is determined by vote stability, and Uniqueness is absolute. Strong GMP also has a \textit{Propagation} condition which forces all core members to attempt to commit the value.

\textbf{Validity} If \( p \) commits \( <v, x> \), then \( p \) is in a subset of \( \text{GpView}^{x-1} \) according to which \( v \) is stable for \( x \):

\[
\text{COMMIT}_p(<v, x>) \Rightarrow \bigvee_{S \subseteq \text{GpView}^{x-1}} \left( p \in S \right) \land \text{STABLEVOTES}_S(<v, x>)
\]

\textbf{Uniqueness} If \( p \) commits \( <v, x> \), no other process, \( q \), ever commits \( <v', x> \), for \( v' \neq v \):

\[
\text{COMMIT}_p(<v, x>) \Rightarrow \bigwedge_{v' \neq v} \bigwedge_{q \in \text{Proc}} \neg \text{COMMIT}_q(<v', x>)
\]
Eventual Propagation If \( p \) commits \( < v, x > \), then every other process \( q \) either
1) is not in \( \text{GpView}^z \), or 2) eventually commits \( < v, x > \), or 3) eventually
fails:

\[
\operatorname{COMMIT}_p(< v, x >) \Rightarrow \\
\bigwedge_{q \in \text{Proc}} \left( \left( \text{OUT-GP}(q) \right) \lor \left( \Diamond \text{COMMIT}_q(< v, x >) \right) \lor \left( \Diamond \text{DOWN}_q \right) \right)
\]

From the discussions in Chapter 5, it is easy to see that Uniqueness cannot be
guaranteed without additional restrictions on stability.

Definition \( < v, x > \) is committably stable (c-stable) if and only if \( < v, x > \) is stable
with respect to a majority subset of \( \text{GpView}^{z-1} \). The formula \( \text{C-STABLE}(< v, x >) \)
holds exactly when \( < v, x > \) is c-stable :

\[
\text{C-STABLE}(< v, x >) \overset{\text{def}}{=} \bigvee_{S \in \text{Maj}(\text{GpView}^{z-1})} \left( \text{STABLEVOTE}_S(< v, x >) \right).
\]

Uniqueness and Validity then combine to restate the latter as

\[
\operatorname{COMMIT}_p(< v, x >) \Rightarrow \bigvee_{S \in \text{Maj}(\text{GpView}^{z-1})} \left( p \in S \right) \land \text{C-STABLE}_S(< v, x >).
\]

6.2 Knowledge Analyses

These analyses use two notions of minimality: a commit protocol is knowledge-
minimal if processes commit a value as soon as they know it is safe to do so; it
is message-minimal if it is impossible to commit a value safely in fewer messages.
In this section, we show that parts of the s-GMP protocol are knowledge-minimal, and how the knowledge-based formulation of Strong GMP led to optimizations in the other parts. We also derive a lower bound on the number of messages required to solve Strong GMP, and compare s-GMP with a message-optimal protocol. The optimizations to s-GMP are themselves minimal in both senses.

For these purposes, the most important aspect of s-GMP's correctness is that at most one value attains c-stability for any given version, from which it follows that a core member may safely commit \(< v, x >\) as soon as \(< v, x >\) becomes c-stable. On the other hand, the following derivation from Validity shows that every solution to Strong GMP necessarily satisfies

\[
\text{COMMIT}_p(\langle v, x \rangle) \Rightarrow K_p \text{C-STABLE}(\langle v, x \rangle).
\]

1. Restating Validity:

\[
\text{COMMIT}_p(\langle v, x \rangle) \Rightarrow \bigvee_{S \in \text{Maj}(\text{GpView}^{x-1})} \left( p \in S \right) \land \text{C-STABLE}_S(\langle v, x \rangle)
\]

2. Applying the Knolwedge Generalization Inference Rule, we obtain:

\[
K_p \text{COMMIT}_p(\langle v, x \rangle) \Rightarrow K_p \left( \bigvee_{S \in \text{Maj}(\text{GpView}^{x-1})} \left( p \in S \right) \land \text{C-STABLE}_S(\langle v, x \rangle) \right).
\]

3. Because \(\text{COMMIT}_p(\langle v, x \rangle)\) is local to \(p\), \(\text{COMMIT}_p(\langle v, x \rangle)\) is equivalent to \(K_p \text{COMMIT}_p(\langle v, x \rangle)\). Substituting gives:

\[
\text{COMMIT}_p(\langle v, x \rangle) \Rightarrow K_p \text{C-STABLE}_S(\langle v, x \rangle). \quad (6.1)
\]
Formula 6.1 states what must be true when a process commits a value. In this regard, the ‘earliest’ a process may safely commit a value is as soon as it knows that value is c-stable:

$$K_p C \text{-STABLE}(<v, x>) \Rightarrow \text{COMMIT}_p(<v, x>). \quad (6.2)$$

We call a commit protocol in which processes commit upon attaining a single ‘level’ of knowledge of c-stability, a 1K-commit protocol.

In what follows we use \( \text{com}(v) \) for any commit message (\( \text{M-com}(v) \) or \( \text{R-com}(v) \)), and \( \text{sub}(v) \) for any submit message.

**Definition** Let \( p' \) send (or multicast) message \( m \), and let \( \text{ack}(m) \) denote a message sent by a recipient of \( m \), back to \( p' \) acknowledging receipt of \( m \). Define

- \( \text{Recipients}(p', m) \triangleq \{ p | \text{RECV}_p(p', m) \} \),
- \( \text{AcksSent}(p', m) \triangleq \{ p | \text{SEND}_p(p', \text{ack}(m)) \} \), and
- \( \text{AcksRcvd}(p', m) \triangleq \{ p | \text{RECV}_{p'}(p, \text{ack}(m)) \} \)

Process failures and message asynchrony render

$$\text{AcksRcvd}(p', m) \subseteq \text{AcksSent}(p', m) \subseteq \text{Recipients}(p', m),$$

for any \( p \) and \( m \). This is significant in the next definition, and in the propositions that follow. To describe the most general, observable system state from which it can be inferred that a value is c-stable, we use these process sets to define ‘successful initiator’ with the following formula:
Definition  Process $p'$ is successful for $<v,x>$ if and only if a majority subset of \( \text{GpView}^{x-1} \) acknowledge $p'$'s submission:

\[
\text{SUCCESSFUL}_{p'}(<v,x>) \overset{\text{def}}{=} \text{AcksSent}(p', \text{sub}(v)) \in \text{Maj}(\text{GpView}^{x-1}). \quad (6.3)
\]

Equation 6.3 captures exactly the problems posed by invisible commits. It leaves open whether an initiator actually received any of the acknowledgements sent to it, which determines whether the initiator were in fact able to commit the update. However, in the absence of concrete evidence, it is impossible for a subsequent reconfigurer to know whether a previous initiator succeeded in committing the update anywhere. As in the behavioral specification of Strong GMP, Epistemic Uniqueness and Eventual Propagation force a reconfigurer to assume an update was committed if it determines the update could possibly have been committed, or, equivalently, if it does not know the previous initiator was not successful. Thus for initiators $r$ and $r'$, $r'$ propagates $r$'s submission $<v,x>$ exactly when

\[-K_{r'} \neg \text{SUCCESSFUL}_{r}(<v,x>).\]

6.2.1 Facts from Correctness

We restate the following facts about s-GMP, proven in Chapter 5

Fact 6.2.1 In s-GMP, if $r$ is successful for $<v,x>$, no value unequal to $v$ is thereafter submitted for version $x$:

\[
\text{SUCCESSFUL}_r(<v,x>) \Rightarrow \bigwedge_{v' \neq v} \bigwedge_{p', p} \neg \text{SEND}_{p'}(p, \text{sub}(<v',x>)).
\]
Fact 6.2.2 In S-GMP, if \( r \) is successful for \( \langle v, x \rangle \), then \( \langle v, x \rangle \) is \( c \)-stable:

\[
\text{SUCCESSFUL}_r(\langle v, x \rangle) \Rightarrow \text{C-STABLE}(\langle v, x \rangle).
\]

Fact 6.2.3 In S-GMP, if \( \text{VOTE}_p(\langle v, x \rangle) \) holds for a majority of \( \text{GpView}^{x-1} \) along any consistent cut, then \( \text{C-STABLE}(\langle v, x \rangle) \) holds along that cut:

\[
\left( \bigvee_{S \in \text{Maj}(\text{GpView}^{x-1})} \left( \bigwedge_{p \in S} \text{VOTE}_p(\langle v, x \rangle) \right) \right) \Rightarrow \text{C-STABLE}(\langle v, x \rangle).
\]

6.2.2 Knowing Stability

To understand when a process executing S-GMP knows a value is \( c \)-stable, we analyze the protocol’s communication phases. We show two communication phases are necessary when the initiator is \( \text{mgr} \). We begin these analyses by considering only those instances of S-GMP when a given process is either the first \( \text{mgr} \), or has been \( \text{mgr} \) for at least one completed update of the system view; the issues arising in the transition of a process from reconfiguration initiator to \( \text{mgr} \) are more complex and we discuss them later.

We also show that in four of the five possible global states that may exist at the start of reconfiguration (\( i.e. \) the degree and type of local view divergence), processes learn \( c \)-stability earlier than when they commit in S-GMP, allowing us, in two cases, to eliminate a full phase of communication in the reconfiguration algorithm, in another case to eliminate the subsequent submit message once the reconfigurer becomes \( \text{mgr} \), and to preclude a fourth situation entirely. We show it is impossible to improve the fifth.
Definition Let ISMGR(p', x) hold if a majority of GpView_\mathcal{x}^{x-1} believe p' is the highest-ranked, non-faulty process. Then version x has a clean starting point if x = 1, or if ISMGR(p', x - 1) held throughout the formation of GpView_\mathcal{x}^{x-1}.

By inspecting s-GMP, we obtain:

Fact 6.2.4 In s-GMP, if x has a clean starting point, and if mgr submits < v, x >, then no process in GpView_\mathcal{x}^{x-1} has previously voted for < v, x >.

The following theorem shows that, in s-GMP, mgr does not learn c-stability of a value before the end of the first phase. In particular, it shows that mgr learns < v, x > is c-stable exactly when it learns it is successful for < v, x >.

Theorem 6.2.1 In s-GMP,

\[ K_{mgr} \text{C-STABLE}(< v, x >) \Leftrightarrow \text{AcksRcvd}(mgr, M-sub(v)) \in \text{Maj}(GpView_\mathcal{x}^{x-1}) \]

which is equivalent to

\[ K_{mgr} \text{C-STABLE}(< v, x >) \Leftrightarrow K_{mgr} \text{SUCCESSFUL}_{mgr}(< v, x >). \]

Proof "⇒" We show \( K_{mgr} \text{VOTE}_p(< v, x >) \Rightarrow p \in \text{AcksRcvd}(mgr, M-sub(v)), \) by inspecting s-GMP. First, we show that mgr learns \( \text{VOTE}_p(< v, x >) \) upon receipt of p's acknowledgement to \( M-sub(v), \) and then that mgr cannot learn \( \text{VOTE}_p(< v, x >) \) unless it receives p's acknowledgement.

Building on [CM86], Mazer [Maz90] showed that if p learns \( \phi_q, \) a formula local to q, and if processes a) are asynchronous, or b) can experience crash failures and \( \phi_q \) can only be made true by q, then p must receive a message from (or indirectly
from) \( q \) that implies \( \phi_q \). \( \text{VOTE}_q(\langle v, x \rangle) \) is such a formula, and our system is both asynchronous and subject to process failures.

Since \( x \) has a clean starting point, no process in \( \text{GpView}^{x-1} \) had voted for \( \langle v, x \rangle \) before receiving \( \text{M-sub}(v) \) (Fact 6.2.4). Inspecting S-GMP shows that a process votes after receiving \( \text{M-sub}(v) \) and before responding to it, and since \( \text{VOTE}_p(\langle v, x \rangle) \) has not held previously, it holds for the first time in the execution immediately before \( p \) acknowledges \( \text{M-sub}(v) \). Thus, \( \text{mgr} \) can infer \( \text{VOTE}_p(\langle v, x \rangle) \) upon receipt of \( p \)'s response to \( \text{M-sub}(v) \).

Finally, \( \text{mgr} \) cannot learn \( \text{VOTE}_p(\langle v, x \rangle) \) from some other process. Outer processes do not send messages to one another\(^1\), so no other process can have learned \( \text{VOTE}_p(\langle v, x \rangle) \) independently, and propagated that information to \( \text{mgr} \). As a result, \( \text{mgr} \) cannot learn \( \text{VOTE}_p(\langle v, x \rangle) \) from a process other than \( p \). The only message \( p \) sends after voting is \( \text{ack}(\text{M-sub}(v)) \), so

\[
K_{\text{mgr}} \text{VOTE}_p(\langle v, x \rangle) \Rightarrow p \in \text{AcksRcvd}(\text{mgr}, \text{M-sub}(v)).
\]

Let \( \text{VOTE}_S(\langle v, x \rangle) \) denote \( \bigwedge_{p \in S} \text{VOTE}_p(\langle v, x \rangle) \). Then,

\[
K_{\text{mgr}} \text{C-STABLE}(\langle v, x \rangle) \Rightarrow \bigvee_{S \in \text{Maj}(\text{GpView}^{x-1})} K_{\text{mgr}} \text{C-STABLE}_S(\langle v, x \rangle) \Rightarrow \\
K_{\text{mgr}} \text{VOTE}_S(\langle v, x \rangle) \Rightarrow S \subseteq \text{AcksRcvd}(\text{mgr}, \text{M-sub}(v)) \Rightarrow \\
\text{AcksRcvd}(\text{mgr}, \text{M-sub}(v)) \in \text{Maj}(\text{GpView}^{x-1}).
\]

\(^1\)Outer processes do not send messages to each other except during reconfiguration. In that case, if \( p \) has sent any message to \( q \) (whether \( p \) is initiating or responding), \( q \) will not send a message to \( \text{mgr} \); \( \text{mgr} \) has been isolated.
"\leq\) Since \(\text{AcksRcvd}(mgr, M\text{-}sub(v)) \subseteq \text{AcksSent}(mgr, M\text{-}sub(v))\), \(mgr\) knows it is successful for \(<v, x>\). Using Fact 6.2.2, \(K_{mgr} C\text{-}\text{STABLE}(v, x)\).

Theorem 6.2.1 shows that whenever \(x\) has a clean starting point \(mgr\) cannot know \(c\)-stability of any value until after collecting responses. We use this in Theorem 6.2.2 to show that \(s\text{-GMP}\), during a \(mgr\)-initiated update begun from a clean starting point, is knowledge-minimal.

**Theorem 6.2.2 (Two-Phase Necessity)** When version \(x\) has a clean starting point, \(s\text{-GMP}\) is knowledge-minimal. That is,

1. \(\text{SEND}_p(mgr, \text{ack}(M\text{-}sub(v))) \Rightarrow \neg K_p C\text{-}\text{STABLE}(v, x)\)

2. \(\text{RECV}_p(mgr, M\text{-}com(v)) \Rightarrow K_p C\text{-}\text{STABLE}(v, x)\).

**Proof** Since \(x\) has a clean starting point, no process voted for \(<v, x>\) before receiving \(M\text{-}sub(v)\). Theorem 6.2.1 and inspecting the protocol show that \(mgr\) does not know \(C\text{-}\text{STABLE}(v, x)\) until after it has sent all \(M\text{-}sub(v)\) messages\(^2\); therefore, \(M\text{-}sub(v)\) cannot have implied \(C\text{-}\text{STABLE}(v, x)\). Furthermore, neither \(p\)'s internal voting event nor its acknowledgement add to its knowledge [CM86], establishing (1).

Again, except during reconfiguration, outer processes do not communicate among themselves so \(p\) cannot have learned \(C\text{-}\text{STABLE}(v, x)\) independently or from another outer process since receiving \(M\text{-}sub(v)\). Because \(p\) does receive \(mgr\)’s

\(^2\)Indeed, under the assumptions, \(C\text{-}\text{STABLE}(v, x)\) does not hold until a majority have received the \(M\text{-}sub(v)\) message. This need not be the case when \(x\) does not have a clean starting point.
\textbf{M-com}(v) message, it has neither sent nor received an \textbf{R-int()} message. In consequence, the earliest \(p\) can learn \textbf{C-STABLE}(< v, x >) is upon receipt of \(mgr\)'s second phase (commit) message. Since \textbf{S-GMP} is full-information, \textbf{C-STABLE}(< v, x >) is propagated by \textbf{M-com}(v).

\section{Optimizations to \textbf{S-GMP}}

This section presents optimizations made possible by a knowledge-theoretic analysis of the degree of local view inconsistency that may exist once \(mgr\) is believed faulty.

\textbf{Fact 6.2.5} Let \(r\) be a reconfiguration initiator with local version \(x - 1\). Then in \textbf{S-GMP}, when collecting interrogation responses exactly one of the following three scenarios is possible:

\begin{itemize}
  \item[(a)] \(r\) learns some process has local version \(x\);
  \item[(b)] \(r\) learns some process has local version \(x - 2\);
  \item[(c)] all processes from which \(r\) receives responses have local version \(x - 1\).
\end{itemize}

In the first instance \(r\) learns \textbf{C-STABLE}(< v, x >) at the end of reconfiguration Phase I. To optimize \textbf{S-GMP} \(r\), instead of proposing < v, x > and collecting responses, commits < v, x > and immediately propagates \textbf{C-STABLE}(< v, x >) by multicasting \textbf{R-com}(v) to any process whose acknowledgement also indicated local version \(x - 1\). By omitting the second phase message, \textbf{R-sub}(v), the optimization saves approximately \(2 \times | \text{GpView}^{x-1} | \) messages.
Another obvious optimization precludes case (b). A process with local version \( x - 2 \), upon receiving \( \text{R-int}(v', x - 1) \), learns \( \text{C-STABLE}(v', x - 1) \); this process can safely commit that update, then respond to the reconfigurer with its new version.

The third optimization is more subtle. In (c), if a majority of \( \text{GpView}^{x-1} \) (among the interrogation respondents) also indicate having most-recently voted for the same value, \( r \) learns (through Fact 6.2.3) that value is c-stable. As in the first optimization, reconfiguration Phase II is unnecessary.

Finally, with the third optimization in mind, consider case (a) again. Suppose, as in Figure 6.2.3, that among \( \text{AcksRcvd}(r, \text{R-int}(x - 1)) \), the respondents indicating local version \( x \) constitute a majority subset of \( \text{GpView}^x = \text{GpView}^{x-1} \pm v \). If such a majority subset also indicates having most recently voted for some value,
\(<v_{x+1}, x+1>\), then \(r\) additionally learns \(\text{C-STABLE}(<v_{x+1}, x+1>)\) (Fact 6.2.3). Reconfigurer \(r\) can immediately multicast a \textit{double commit} message, \(\text{R-com}(v_x) : \text{M-com}(v_{x+1})\).

**Proposition 6.2.1 (Optimization 1)** If \(r\) is a reconfiguration initiator with local version \(x-1\), and some process's response to \(\text{R-int}(x-1)\) indicates local version \(x\), then \(r\) learns \(\text{C-STABLE}(<v, x>)\) upon receipt of \(\text{ack}(\text{R-int}(x-1))\) from that process.

**Proof** If some \(p\) sends an \(\text{ack}(\text{R-int}(x-1))\) indicating \(\text{ver}(p) = x\), then \(K_r \text{COMMIT}_p(<v, x>)\). Distributing \(K_r\) over implication in Validity gives the desired \(K_r \text{C-STABLE}(<v, x>)\).

**Proposition 6.2.2 (Optimization 2)** If \(r\) is a reconfiguration initiator with local version \(x-1\), then a process with local version \(x-2\) learns \(\text{C-STABLE}(<v, x-1>)\) upon receiving \(\text{R-int}(x-1)\) from \(r\).

**Proof** Similar to Proposition 6.2.1.

**Proposition 6.2.3 (Optimization 3)** Let \(r\) be a reconfiguration initiator with local version \(x-1\), and suppose all processes responding to \(\text{R-int}(x-1)\) indicate local version \(x-1\). If some subset of \(\text{AcksRcvd}(r, \text{R-int}(x-1))\) are a majority subset of \(\text{GpView}^{x-1}\) and report identical (non-null) \textbf{NextUpdate} values, then \(r\) learns that value is c-stable.
Proof Follows immediately from Fact 6.2.3.

Proposition 6.2.4 (Optimization 4) Let $r$ be a reconfiguration initiator with local version $x - 1$. Let $S$ be the subset of $\text{AcksRcvd}(r, R-\text{int}(x - 1))$ reporting local version $x$, and suppose $\text{LocalView}^{r}_{S}$ is $\text{LocalView}^{r-1}_{r}$ altered by $v$; $\text{LocalView}^{r}_{S} = \text{LocalView}^{r-1}_{r} \pm v$. Let $v_1$ and $v_2$ (both possibly null) be the values the members of $S$ report having most recently voted for to form $\text{GpView}^{r+1}$, with $S_1$ the subset of $S$ that voted for $v_1$, and $S_2$ the subset of $S$ that voted for $v_2$. Then for $1 \leq i \leq 2$,

$$S_i \in \text{Maj}(\text{LocalView}^{r}_{S}) \Rightarrow K_r C-\text{STABLE}(< v_i, x + 1 >).$$

Proof Proposition 6.2.1 shows $r$ learns $C-\text{STABLE}(< v, x >)$ at the end of Phase I. Upon committing $v$ to form $\text{LocalView}^{r}$ and then determining whether either of the $S_i$ constitute a majority subset of $\text{LocalView}^{r}$, applying Fact 6.2.3 gives the final result.

The next theorem shows $S$-GMP, as originally presented, is knowledge-minimal (i.e. cannot be optimized) when all respondents to $R-\text{int}(x - 1)$ report the same local version, but no majority subset has most-recently voted for the same value.

Theorem 6.2.3 (Three-Phase Necessity) Let $r$ be a reconfiguration initiator with local version $x - 1$, and suppose all processes in $\text{AcksRcvd}(r, R-\text{int}(x - 1))$ indicate local version $x - 1$. If, among $\text{AcksRcvd}(r, R-\text{int}(x - 1))$, no majority subset of $\text{GpView}^{r-1}$ further indicate having most-recently voted for the same value, then for all $< v, x >$ and $p \in \text{AcksRcvd}(r, R-\text{int}(x - 1))$,
1. \( \text{SEND}_p(r, \text{ack}(R\text{-sub}(v))) \Rightarrow \square \neg K_p \text{C-STABLE}(<v, x>) \)

2. \( \text{RECV}_p(r, R\text{-com}(v)) \Rightarrow K_p \text{C-STABLE}(<v, x>) \).

**Proof** Upon initiating reconfiguration, \( r \) does not know whether any value is c-stable for version \( x \); it has not received \( M\text{-com}(x) \) from the previous \( mgr \), has not learned \( \text{C-STABLE}(<v, x>) \) from a previous reconfigurer, and has not received messages from non-initiator processes. Its interrogation cannot imply \( \text{C-STABLE}(<v, x>) \), and clearly \( p \)'s response to \( r \) does not add to \( p \)'s knowledge.

Since no majority of \( r \)'s respondents voted for the same value, \( r \) cannot distinguish at the end of Phase I which, if any, of the reported pending values may be c-stable; \( r \) can envision scenarios in which each of the reported values are c-stable. S-GMP correctness only ensures that if a value is c-stable, \( r \) will propose it; i.e., if \( <v', x>\) and \( <v, x>\) are values \( r \)'s respondents report pending, correctness and \( K_r \neg \text{C-STABLE}(<v', x>) \) do not imply \( <v, x> \) is c-stable, so \( R\text{-sub}(v) \) cannot imply \( \text{C-STABLE}(<v, x>) \). Since outer processes have not sent messages among themselves, none can have learned c-stability before receiving \( R\text{-sub}(v) \), and neither \( p \)'s vote nor its response add to its knowledge.\(^3\)

As in Theorem 6.2.2, \( r \) learns \( \text{C-STABLE}(<v, x>) \) if \( \text{AcksRcvd}(r, R\text{-sub}(x)) \) is a majority subset of \( \text{GpView}^{r-1} \), which outer processes learn from \( r \)'s commit message.

---

\(^3\)Proposing \( <v, x> \) and an outer process receiving the proposal does not guarantee \( <v, x> \) will become c-stable. Two failure scenarios in the second and third phases of reconfiguration (\( r \) failing before sending all proposal messages, or a majority failing before receiving and/or voting for the proposal) result in \( <v, x> \) not becoming c-stable. Moreover, both \( r \) and the outer processes envision these scenarios.
6.3 Optimized S-GMP

Optimized S-GMP alters S-GMP by the four optimizations just outlined. These affect only the Reconfigurer task, and its subroutine, DetermineProposal.

Task: Optimized Reconfiguration Initiator, \( r \); \( \text{ver}(r) = x - 1 \).

\[
\text{multicast}_r(\text{LocalView}_r, \text{R-int}(x - 1));
\]

for all \( p \in \text{LocalView}_r \)

\[
\text{recv}_r(p, \text{NextUpdate}_p : \text{LastCommit}_p) \text{ or } \text{faulty}_r(p);
\]

if (majority of \( \text{LocalView}_r \) didn’t respond) \( \text{quit}_r \);

DetermineProposal(\( v1 \), ver, \( v2 \));

if (OptimizedFlag) /* Omit Phase II. */

DoPrecommit(\( v1 \));

DoCommit(\( v1 \));

DoPrecommit(\( v2 \));

if (DoubleCommitFlag) /* Commit \( < v2, \text{ver+1} > \), too. */

DoCommit(\( v2 \));

\[
\text{multicast}_r(\text{LocalView}_r, \text{R-com}(< v1, \text{ver} >) : \text{M-com}(< v2, \text{ver+1} >));
\]

\( \text{mgr}, \text{v1} = r, \bot; \)

else

\[
\text{multicast}_r(\text{LocalView}_r, \text{R-com}(< v1, \text{ver} >) : \text{M-sub}(< v2, \text{ver+1} >));
\]

\( \text{mgr}, \text{v1} = r, v2; \)

goto Exit;
Optimized Reconfigurer, Continued.

/* No optimization possible. Must do 3-Phase. */

else

    DoPreCommit(v1);

    multicast_r(LocalView_r, R-sub(< v1, ver >));

    forall p \in LocalView_r

        recv_r(p, ack(R-sub(< v1, ver >))) or faulty_r(p);

    if (majority of LocalView_r didn’t respond) quit_r;

    DoCommit(v1);

    multicast_r(LocalView_r, R-com(< v1, ver >): M-sub(< v2, ver + 1 >));

    mgr, v1 = r, v2:

Label: Exit

Begin mgr Task:
/* Optimized DetermineProposal for reconfigurer, r. */

/* ver(r) = x - 1. */

Procedure: DetermineProposal(OUT proposal, OUT version, OUT invisible);

$\text{Ahead}_r = \{[\text{ld}s, \text{ver}(p)]_p \mid \text{ver}(p) = x\}$;

/* Behind$_r \neq \emptyset$ is no longer possible! */

$\text{SubAhead}_r = \{[\text{ld}s, \text{ver}(p) + 1, \text{rank}(\text{init})]_p \mid \text{ver}(p) = x = \text{ver}(r) + 1\}$;

$\text{SubCurrent}_r = \{[\text{ld}s, \text{ver}(p) + 1, \text{rank}(\text{init})]_p \mid \text{ver}(p) = x - 1 = \text{ver}(r)\}$;

version = $x$;

if ($\text{Ahead}_r \neq \emptyset$)

    set OptimizedFlag;

    proposal = $\text{Ahead}_r$;

    Let $\text{LocalView}^x_r = \text{LocalView}^{x-1}_r \pm \{\text{proposal}\}$;

    GetStableProposal(invisible, SubAhead$_r$);

    DoubleCommitFlag = $\left(\{p \mid \text{NextUpdate}_p \Rightarrow \text{invisible}\} \in \text{Maj}(\text{LocalView}^x_r)\right)$;

return();

/* DetermineProposal Continues on next page */
Optimized DetermineProposal, continued.

    /* All respondents report the same local version, x - 1. */

    if (SubCurrent_r == 0)
        proposal = [-mgr, x, rank(r)];
    GetUpdate(invisible);
    OptimizedFlag, DoubleCommitFlag = 0, 0;
    return();

    if (SubCurrent_r is a singleton)
        proposal = SubCurrent_r;
        OptimizedFlag = \left\{ p \mid NextUpdate_p == proposal \right\} \in \text{Maj}(LocalView_r^{x-1})
    GetUpdate(invisible);
    DoubleCommitFlag = 0;
    return();

    /* SubCurrent_r has (exactly) two elements. */

    GetStableProposal(proposal, SubCurrent_r);
    OptimizedFlag = \left\{ p \mid NextUpdate_p == proposal \right\} \in \text{Maj}(LocalView_r^{x-1})
    GetUpdate(invisible);
    DoubleCommitFlag = 0;
    return();
6.4 Message-Minimality

We use Mazer’s Message Chain Theorem [Maz90] to determine the minimal number of messages required by any solution to Strong GMP. Similar work for different commit-style problems is in [Had91] and [MF90]. For simplicity, we assume no process has knowledge of another’s votes.

Recall that $\text{Sizeof-Maj}(S)$ is the size of the smallest majority subset of $S$. We show that, for a given set $S$, at least $2(\text{Sizeof-Maj}(S) - 1)$ messages are necessary for any member of $S$ to learn $c$-stability of a value, and that any algorithm using fewer than $(|S| - 1) + 2(\text{Sizeof-Maj}(S) - 1)$ messages does not satisfy Eventual Propagation.

A process can safely commit a value as soon as it knows it to be $c$-stable, and we know that any solution to Strong GMP is necessarily a 1$K'$-commit protocol. Let $G \in \text{Maj}(S)$ and let $C$-$\text{STABLE}_G(<v, x>)$ hold. Then $K_{p'} C$-$\text{STABLE}(<v, x>)$ if and only if

$$\bigwedge_{p \in G} \left( K_{p'} \text{VOTE}_p(<v, x>) \right) \land$$

$$\bigwedge_{p \in G} \bigwedge_{v' \neq v} K_{p'} \left( \Box \neg \text{VOTE}_p(<v', x>) \right) \land \bigwedge_{q \in G} K_{p'} D_G \text{FAULTY}(q).$$

Since $\text{VOTE}_p(<v, x>)$ is local to $p$, $p'$ must receive (at least) one message before $K_{p'} \text{VOTE}_p(<v, x>)$ can hold, so at least $\text{Sizeof-Maj}(S) - 1$ messages are needed to satisfy the first conjunct. For the second conjunct, nothing in the specification of Strong GMP ensures an initiator that an outer process’s vote for $<v, x>$ is stable.

One solution is initial stability - each process votes for one and only one value - but then there are many initial configurations in which no majority concurs on
a specific value\textsuperscript{4}. Alternatively, if we allow processes to vote for more than one value over time, $p'$ must somehow learn stability. It is clear that processes cannot independently (\textit{i.e.} without communicating) decide when to stabilize a vote, as this degenerates to initial stability. Therefore the decision to stabilize a vote must be coordinated to ensure that a majority stabilize a particular value, and this requires at least $\text{Sizeof-Maj}(S) - 1$ more messages, giving a total of $2(\text{Sizeof-Maj}(S) - 1)$ messages before $p'$ can commit.

There are two possible patterns of communication to achieve this:

- $p'$ can passively collect votes until a single value has been voted for by a majority. At this point it sends a ‘stabilize’ message to each process in the majority subset. Note that $p'$ cannot commit here since it does not yet know whether each outer process has stabilized\textsuperscript{5}. This requires an additional $\text{Sizeof-Maj}(S) - 1$ messages, totaling $3(\text{Sizeof-Maj}(S) - 1)$ messages, before $p'$ can commit.

- $p'$ can choose a value and actively propagate it to a majority subset. Upon receipt of the value, each process votes and stabilizes, then sends confirmation of these actions back to $p'$. This approach requires a minimum of $2(\text{Sizeof-Maj}(S) - 1)$ messages for $p'$ to learn $c$-stability.

Finally, Eventual Propagation requires that once $p'$ commits a value, every functional process must also commit that value. If every functional process undertakes to learn $c$-stability in the fashion outlined above, a minimum of $\lfloor |S| -$

\textsuperscript{4}This is not true for $|V| \leq 2$.

\textsuperscript{5}A technicality now, but important in practice when reconfiguration may be ongoing.
$f \times 2(\text{Sizeof-Maj}(S) - 1)$ total messages are needed (where $f$ is the number of crashed processes). On the other hand, if $p'$, having learned $c$-stability 'the hard way', propagates this fact, only an additional $|S| - 1$ messages are needed\(^6\), giving a minimal total of $(|S| - 1) + 2(\text{Sizeof-Maj}(S) - 1)$ messages.

In s-GMP, installing version $x$ from a clean starting point and assuming $mgr$ is not thought faulty, uses at most $3(|\text{GpView}^{x-1}| - 1) - f$ messages. The extra messages (approximately $|\text{GpView}^{x-1}| - f - 1$) represent the tradeoff between compressing $mgr$'s faultiness determinations to one phase, and spacing them out over, in the worst case, $\text{Sizeof-Maj}(\text{GpView}^{x-1}) + 1$ waiting periods. This case would occur when $p'$ chooses the worst possible sequence of processes from which to get responses: if, from the initial majority subset it chooses, $p'$ receives only $\text{Sizeof-Maj}(\text{GpView}^{x-1}) - 2$ responses, then observes $\left\lceil \frac{|\text{GpView}^{x-1}|}{2} \right\rceil - 2$ successive 'failures' before getting the last response.

### 6.5 Versions Without Clean Starting Points

Finally, we consider the case when a version does not have a clean starting point. Recall that version $x$, with $p'$ as $mgr$, has a clean starting point exactly when $x = 1$ or $\text{ISMGR}(p', x - 1)$ held throughout the formation of $\text{GpView}^{x-1}$. By definition then, version $x$ does not have a clean starting point exactly when reconfiguration were involved in installing $\text{GpView}^{x-1}$. To conclude this section, we show optimized s-GMP is knowledge minimal irrespective of its starting point.

\(^6\)The lack of $f$ here arises from the impossibility of $p'$ knowing whether another process is truly crashed.
Corollary 6.5.1 In Optimized S-GMP, reconfigurer $r$ only proposes and commits version $\text{ver}(r) + 1$. If the Double Commit optimization were applicable, $r$ also commits $\text{ver}(r) + 2$.

**Proof** From Proposition 6.2.2, Optimization 2 ensures $\text{Behind}_r$ is null, so $r$ does not propose version $\text{ver}(r)$ (or any version less than $\text{ver}(r)$). It also cannot propose version $\text{ver}(r) + 3$, as every process with local version $\text{ver}(r) + 2$ believes $r$ faulty before it can receive $r$’s $\text{R-int}(x - 1)$.

Suppose $\text{GpView}^{x-1}$ were installed via reconfiguration. Let $r$ be the initiator that completed it, $v_{x-1}$ be the value committed in forming $\text{GpView}^{x-1}$, and $v_x$ be the value $r$, as the new $\text{mgr}$, submits and commits for $\text{GpView}^x$. We prove no member of $\text{GpView}^{x-1}$ knows $\text{C-Stable}(<v_x,x>)$ before it receives $r$’s Phase II message, $\text{M-com}(<v_x,x>)$.

From Corollary 6.5.1, at the outset of its reconfiguration attempt, $\text{ver}(r) = x - 2$. The responses $r$ receives to $\text{R-int}(x - 2)$ fall into one of the following classes:

**State Unity** All $p$ responding to $\text{R-int}(x - 2)$ reported $\text{LastCommit}_p = [v_{x-2}, x - 2]$, and either $\text{NextUpdate}_p = [v, x - 1, \text{rank}()]$ for some value $v$, or $\text{NextUpdate}_p = [-\text{mgr}, x - 1, \text{rank}()]$. Reconfigurer $r$ determines $v_x = (v \oplus -\text{mgr})$ according to the minimal value among the $\text{rank}()$s reported, or if Optimization 3 (Proposition 6.2.3) is applicable, by majority reporting.

**State Divergence** Some subset of $\text{LocalView}^{x-2}$ reported $\text{LastCommit} = [v_{x-1}, x - 1]$, and $p'$ is the core member with least rank among those reported to $r$ to
have made a submission in \textit{Submissions}_r(x).

If \( r \) detects State Unity at the end of reconfiguration Phase I, it eventually multicasts \( \text{R-com}(<v_{x-1}, x-1>) \) resulting in \( \text{GpView}^{r-1} \), followed by (piggybacked or not) \( \text{M-sub}(<v_x, x>) \), its proposal for \( \text{GpView}^r \).

\textbf{Theorem 6.5.1} Optimized s-gmp, in the situation described by State Unity, is knowledge-minimal.

\textbf{Proof} We show no functional process in \( \text{GpView}^{r-1} \) infers \( \text{C-stable}(<v_x, x>) \) before receiving \( \text{M-com}(<v_x, x>) \) from \( r \).

By hypothesis \( \text{GpView}^{r-1} = \text{LocalView}_r^{r-2} \pm v_{x-1} \) and all \( r \)'s Phase I respondents report local version \( x-2 \). Clearly, no process in \( \text{LocalView}_r^{r-2} \) knows of \( v_x \) when it responds to \( \text{R-int}(x-2) \). There are two cases to consider:

1. The commit message \( \text{R-com}(<v_{x-1}, x-1>) \) has no contingent update.

In this case \( r \) determines \( <v_x, x> \) only after becoming \textit{mgr}. No process of \( \text{GpView}^{r-1} \) knows of the proposal \( <v_x, x> \) until \( r \) as \textit{mgr} multicasts it. We can apply the proof of Theorem 6.2.2.

2. The commit message is of the form \( \text{R-com}(<v_{x-1}, x-1>):\text{M-sub}(<v_x, x>) \).

(a) If \( <v_{x-1}, x-1> \) is the addition of \( v_{x-1} \) to \( \text{GpView}^{r-2} \), it may be possible that the members of \( v_{x-1} \) knew \( \text{C-stable}(<v_x, x>) \) before \( r \) multicasted \( \text{M-com}(<v_x, x>) \). This could happen if an initiator other than \( r \), say \( r_0 \), had previously sent \( \text{R-com}(<v_{x-1}, x-1>):\text{M-sub}(<v_x, x>) \) and the subsequent \( \text{M-com}(<v_x, x>) \) to the members of \( v_{x-1} \).
Assuming State Unity, every member of $\text{AcksRcvd}(r, R-int(x - 2))$ reported local version $x - 2$. By Gossip, Disconnect and the Initiate condition, none can have responded to $r_0$'s $M$-$\text{sub}(<v_x, x>)$ so $r_0$ would only be able to commit $<v_x, x>$ if $\left(\{r_0\} \cup v_{x-1}\right)$ were a majority subset of $GpView^{x-1}$. If this were true, $r$ would learn it from the responses (from the members of $v_{x-1}$) to $r$'s Join and State-Xfer messages, at which point $r$ would multicast the Double Commit message $R$-$\text{com}(<v_{x-1}, x - 1>); M$-$\text{com}(<v_x, x>)$ (as per Optimization 4, Proposition 6.2.4). Since the only communication is between initiators and outer processes, the initiator is the channel through which knowledge of other members' local states flows. No member of $v_x$ knows whether its cohorts acknowledged $r_0$'s $M$-$\text{sub}(<v_x, x>)$ so none can independently infer $C$-$\text{STABLE}(<v_x, x>)$.

(b) If $<v_{x-1}, x - 1>$ is the removal of $v_{x-1}$ from $GpView^{x-2}$, the members of $v_{x-1}$ cannot form a majority subset, as above, that would enable $r_0$ to commit $<v_x, x>$; only $r$ can make $<v_x, x>$ c-stable.

Under State Divergence, some subset of $\text{AcksRcvd}(r, R-int(x - 2))$ report local version $x - 1$, as well as a (possibly null) expected next update value for version $x$. The only process that could know $C$-$\text{STABLE}(<v_x, x>)$ without receiving $M$-$\text{com}(v_x)$ from $r$ (either as a Double Commit with $R$-$\text{com}(<v_{x-1}, x - 1>)$, or independently), would be some previous initiator from which $r$ no longer re-
ceives messages, and which, because of its rank, is now isolated from the set \( \text{AcksRcvd}(r, R-\text{int}(x - 2)) \).

### 6.6 Epistemic Quantification of Strong GMP

This section quantifies the epistemic achievements of Strong GMP, as specified in Chapter 3, over a complete system run.

GMP-3 requires every process's local version \( x \) to be the same as every other process's local version \( x \), while GMP-2 says that there must be some point in every execution when \( \text{GpView}^x \) exists, resulting in a causally-constrained consensus. Thus, when \( p \) commits version \( x \) it knows eventually \( \text{LocalView}_p^x \) will be the \( x^{th} \) system view. We formalize this as

\[
(\text{ver}(p) = x) \Rightarrow K_p \Diamond (\text{GpView}^x = \text{LocalView}_p^x)
\]

Define the formula \( \text{ISGpView}(x) \) to hold exactly when \( \text{GpView}^x \) is defined:

\[
\text{ISGpView}(x) \overset{\text{def}}{=} \bigwedge_p \left( \text{DOWN}_p \lor \left( \text{ver}(p) = x \land \bigwedge_q \left( \text{DOWN}_q \lor \left( \text{ver}(q) = x \land \text{LocalView}_p^x = \text{LocalView}_q^x \right) \right) \right) \right)
\]

Let \( c_x \) be any consistent cut along which \( \text{GpView}^x \) is defined. Then (modulo failures)

\[
\text{ISGpView}(x) \Rightarrow \left( \bigwedge_p \text{ver}(p) = x \right) \Rightarrow \left( \bigwedge_p K_p \Diamond \text{ISGpView}(x) \right)
\]
This is not eventual common knowledge\textsuperscript{7} \cite{HM84} of the existence of $\text{GpView}^x$. In essence, our Strong GMP specification is phrased loosely-enough so that processes only know that individual instances of local views must be identical. The specification does not make explicit when the system view comes into existence, only that it does. Just as no process can ever know whether another has failed, neither can a process, upon committing a new local view, know whether another has also done so.

However, our specification and the fact that group views exist in sequence are a source of hindsight about previous system views. This could be phrased as, “when $p$’s local version is greater than $x$, $p$ knows that, at some point in the past, the $x^{th}$ system view existed”:

$$ \left( \text{ver}(p) > x \right) \Rightarrow K_p \diamond \text{IsGpView}(x) $$

Upon receipt of $\text{com}(< v_x, x >)$, process $p$ can reason about previous group views. It knows that other processes, also in $\text{LocalView}_p^x$ and still functioning, received and responded to some $\text{sub}(< v_x, x >)$ (irrespective of the submitter’s identity). Because channels are FIFO, $p$ also knows these processes received $\text{com}(< v_{x-1}, x-1 >)$. That is, when $p$ receives $\text{com}(< v_x, x >)$, $p$ knows $\text{GpView}^{x-1}$ was a defined group view:

$$ \left( \text{ver}(p) = x \right) \Rightarrow K_p \diamond \text{IsGpView}(x-1) \quad (6.4) $$

Equation 6.4 holds along any consistent cut containing $p$’s receipt of $\text{com}(< v_x, x >)$.

The passage of time and existence of successive views gives a process deeper knowl-

\textsuperscript{7}Eventual common knowledge would be $\text{IsGpView}(x) \Rightarrow \bigwedge_p \diamond K_p \text{IsGpView}(x)$.\n
edge of past group views. Then, since \( \text{IsGPView}(x) \Rightarrow \bigwedge_{p} \text{ver}(p) = x \),

\[
(\text{ver}(p) = x) \Rightarrow \left( K_{p} \diamond \bigwedge_{q} (\text{ver}(q) = x - 1) \right) \Rightarrow \\
\left( K_{p} \diamond \bigwedge_{q} K_{q} \diamond \text{IsGPView}(x - 2) \right) \Leftrightarrow \left( K_{p} \diamond E \diamond \text{IsGPView}(x - 2) \right)
\]

Conjoining over all processes, \( p \), we obtain

\[
\text{IsGPView}(x) \Rightarrow \\
\bigwedge_{p} K_{p} \diamond E \diamond \text{IsGPView}(x - 2) \Leftrightarrow E \diamond E \diamond \text{IsGPView}(x - 2)
\]

Equations 6.4 and 6.5 state that processes only have knowledge about each others’
local views after the fact. Unwinding these equations further gives the general
result

\[
\text{IsGPView}(x) \Rightarrow (E \diamond \# \text{IsGPView}(x - y)).
\]

If we assume \textit{mgr} does not fail, we obtain a more sophisticated degree of consen-
sus than the specification requires. When \( p \) receives \( \text{com}(<v_x, x>) \), it knows there
is some consistent cut \textit{that includes its current, local state} (\textit{i.e.} \( p \) need not take
any further steps) along which every other functional process in the group will also
receive \( \text{com}(<v_x, x>) \). That is, \( p \) knows it is ‘sitting on’ a particular consistent
cut, but doesn’t know whether the other processes have reached it yet. This is
precisely formulated by the concurrency operator, \( P_{p} \) [PT88], which differentiates
concurrent knowledge from other epistemic formulations. Formalizing this notion
gives

\[
(\text{ver}(p) = x) \Rightarrow K_{p} P_{p} \text{IsGPView}(x).
\]
As only one group view can exist along any consistent cut, let $G_c$ denote the functional members of the group view that exists along cut $c$. Then the following holds whenever $GpView^\tau$ is defined:

\[(\text{IsGpView}(x) \Rightarrow \bigwedge_{p \in G_c} K_p P_p \text{IsGpView}(x)) \Leftrightarrow (\text{IsGpView}(x) \Rightarrow E_{G_c}^C \text{IsGpView}(x))\]  

(6.5)

Equation (6.5) is the induction rule for Concurrent Common Knowledge, making both the composition and existence of $GpView^\tau$ CCK.8

This is not the case when $mgr$ can fail. When $p$ receives $\text{com}(<v_x,x>)$, it does not know whether that sender failed before completing the multicast. The Strong GMP specification only guarantees $p$ that eventually $GpView^\tau$ will be defined:

\[(\text{ver}(p) = x) \Rightarrow K_p \Diamond \text{IsGpView}(x) \Rightarrow (\text{ver}(p) = x) \Rightarrow K_p \Diamond \left( \bigwedge_{q \in G_c} \text{ver}(q) = x \right) \Rightarrow (\text{ver}(p) = x) \Rightarrow K_p \Diamond \left( \bigwedge_{q \in G_c} (K_q \Diamond \left( \bigwedge_{q' \in G_c} \text{ver}(q') = x \right)) \right) \cdots\]

---

8In the terminology of [Tay90], $c_x$ is a locally-distinguishable consistent cut along which $\text{IsGpView}(x)$ holds, the existence of which suffices for concurrent common knowledge of $\text{IsGpView}(x)$. 

Chapter 7

The Role of Order

This chapter investigates the role and cost of providing order to a Membership Resource Manager. We compare Strong GMP with two weaker GMP versions, neither of which provide Order. Combining both weak versions and Order gives Strong GMP.

We use \textit{Consensus}\textsubscript{S}(c) instead of \textit{GpView}\textsubscript{S}(c) to emphasize that consensus among individual processes' local views differentiates the various Group Membership Problems we will discuss in this chapter.

7.1 Characteristic Membership Properties

The properties described here are a broader version of those stated in Chapter 3. Since we are concerned with comparing only consistency properties, we omit explicit reference to the Liveness and Validity properties.
**Eventual Propagation** If $p$ executes $add_p(r)$ (respectively $remove_p(r)$) along cut $c$, then every process, $q$, in $LocalView_p(c)$ eventually executes $add_q(r)$ (respectively $remove_q(r)$) or fails:

$$ADD_p(r) \Rightarrow \bigwedge_{q \in LocalView_p} \left( \Diamond ADD_q(r) \lor \Diamond DOWN_q \right)$$

$$REMOVE_p(r) \Rightarrow \bigwedge_{q \in LocalView_p} \left( \Diamond REMOVE_q(r) \lor \Diamond DOWN_q \right)$$

Eventual Propagation prevents processes from taking actions unilaterally. In fact, it implicitly forces communication between $p$ and the members of its local view before $p$ alters it; $p$ cannot take an action independently and then hope it can propagate this action to the functional members of its local view.

**Convergence** (with respect to formula $\phi$). If eventually $\phi$ is true in a run, then eventually a consensus view exists:

$$\Diamond \phi \Rightarrow \Diamond \left( \bigvee_{S \subseteq Proc} Consensus_S(c) = S \right)$$

Quiescence ("Hereafter, neither failures nor recoveries are suspected by members of $S$") is a common example:

$$QUIET(S) \overset{\text{def}}{=} \bigwedge_{p \in S} \left( \bigwedge_{q \in S} \Box \neg FAULTY_p(q) \right) \bigwedge_{r \notin S} \Box \neg OPERATING_p(r)$$
Note that Convergence with respect to quiescence is actually a restriction on
the failure suspector’s inaccuracy; since we require a suspector to eventually
detect all true failures and recoveries, quiescence requires it not to suspect
falsely when none are occurring.

**Uniqueness** For all asynchronous runs and for all consistent cuts in these runs,
at most one subset, $S$, of processes satisfies $\text{Consensus}_S(c) = S$.

For all runs, $r$, and all cuts, $c$:

$$\left( \bigwedge_{S \subseteq \text{Proc}} \text{UNDEFINED}(\text{Consensus}_S(c)) \right) \lor
\bigvee_{S \subseteq \text{Proc}} \left( (\text{Consensus}_S(c) = S) \land \bigwedge_{S' \neq S} \text{UNDEFINED}(\text{Consensus}_{S'}(c)) \right)$$

**Order** All processes exhibit the same sequence of local views contemporaneously,
provided the views are defined:

$$\bigwedge_p \bigwedge_{0 \leq x} \bigvee_{q \in \text{LocalView}_p^x} \bigwedge_{i > 0} \left( \text{IN-LOCAL}_{p}^{x+1}(q) \Rightarrow \text{DOWN}_q \lor \left( \text{LocalView}_{q}^{x+1} = \text{LocalView}_{p}^{x+1} \right) \right) \lor
\left( \neg \text{IN-LOCAL}_{p}^{x+1}(q) \Rightarrow \bigwedge_{i > 0} \text{UNDEF'D}(\text{LocalView}_{q}^{x+i}) \right).$$

An equivalent formulation is that processes in a consensus view proceed to
the same next consensus view:

Let $c$ be a consistent cut such that $\text{LocalView}_p(c) = \text{LocalView}_p^x = \text{LocalView}_q^y = \text{LocalView}_q(c)$. Let $p, q \in S$ such that $\text{Consensus}_S(c) = S$. Then

$$\left( \text{LocalView}_{p}^{x+1} = \text{LocalView}_{q}^{y+1} \right) \lor \text{UNDEFINED}(\text{LocalView}_{p}^{x+1})$$
7.2 Three MRM Services

In this section we examine implementing MRM services that ensure

- Convergence with respect to QUIET(S) only,
- Eventual Propagation only,
- all four characteristic properties (Strong GMP).

Using the knowledge-based techniques presented in Chapter 6 we can show that each protocol we discuss is message-minimal for the MRM it implements. Message minimality gives a clean, concrete way of measuring the cost of providing Order. We introduce another important metric: the relative expense applications incur when using the different MRMs. Clearly, strong consistency guarantees provide applications with more information so that the application, itself, need not do as much work to accomplish its task. One would conclude that if Order can be provided cheaply there would be no reason not to do so.

We first show that irrespective of join and leave frequency, an MRM providing Strong GMP has very low overhead compared to the weaker services. Rather surprisingly, we show that when joins and leaves are frequent (as is the case in most system executions) implementing Strong GMP uses fewer messages overall than any of the weaker services because successive phases of S-GMP are compressed. Finally, we show that the Strong GMP guarantees permit larger changes between consensus views than does the Eventual Propagation guarantee.
7.2.1 Virtual Partitions

The Virtual Partitions protocol (hereafter VP) of El Abbadi, et.al. [ASC85], [AT86] is an example of a membership service providing only Convergence with respect to QUIET(S). Virtual Partitions were proposed as approximations of the can-communicate-with relation.\footnote{Can-communicate-with is not assumed transitive.} A process attempts to create a new virtual partition when it detects a discrepancy between (its local view of) the virtual partition to which it currently belongs and the can-communicate-with relation; for example, if it receives a message from some process not in its current virtual partition. The virtual partition to which a process actually belongs may be quite different from the one to which it believes it belongs; we discuss this in greater detail in Section 7.2.2. Moreover, this has important ramifications for distributed applications, and we discuss these in Section 7.4. The VP protocol is a two-phase commit protocol with no minimum approval quota.

Convergence with respect to QUIET(S) requires at least $|\text{Proc}| + 2| S | - 3$ messages, and the VP protocol achieves this with the minimal number of messages when QUIET(S) holds. There is no limit on the number of processes that can be added or removed between successive virtual partitions (local views).

7.2.2 The VP Algorithm

The following is taken from [ASC85], but amended slightly here for clarity. The variable $VPid$ is a monotonically increasing integer. Since Virtual Partitions model changes in the communication topology, the Disconnect and Gossip properties are
irrelevant.

In contrast to s-GMP, an outer member of the VP protocol has no guarantee upon receipt of the commit message, \texttt{VP-join(VPid, NewVP)}, that any process other than itself actually joins the new partition \textit{NewVP}; any process mentioned in \textit{NewVP} may have received a more recent \texttt{VP-invite(VPid')} message, in which \textit{VPid} < \textit{VPid}′ renders \textit{VPid} invalid. Despite adopting \textit{NewVP} as its current virtual partition, a process has no assurance that its cohorts will do likewise. In fact, what a process believes is its virtual partition need never have existed as such throughout the entire execution. The ramifications of this are that a process cannot unilaterally take any action that depends upon its virtual partition actually existing – it must first take some action to ensure that what it believes is its virtual partition exists as such.
Task: Initiator, \textit{init}

\[ VPid = \text{generate-unique-new-VPid}; \]

for each processor \( p \)

\[ \text{send}_{\text{init}}(p, \text{VP-invite}(VPid)); \]

await responses;

\[ NewVP = \text{Acks}(\text{init}, \text{VP-invite}(VPid)); \]

for each \( p \in NewVP \)

\[ \text{send}_{\text{init}}(p, \text{VP-join}(VPid, NewVP)); \]

Task: Outer Processes, \( p \)

\[ \text{recv}_{p}(\text{init}, \text{VP-invite}(VPid)); \]

if (\( VPid > \text{max-VPid-seen} \))

\hspace{1cm} depart current virtual partition;

\[ \text{send}_{p}(\text{init}, \text{ack(VP-invite}(VPid))); \]

\[ \text{recv}_{p}(\text{init}, \text{VP-join}(VPid, NewVP)); \]

if (\( VPid > \text{max-VPid-seen} \)) and (same-initiator))

\hspace{1cm} join \( VPid \) and adopt \( NewVP \);

Figure 7.1: The Virtual Partitions Algorithm.
7.2.3 Eventual Propagation Only

We also devised a message-minimal Eventual-Propagation-only protocol (hereafter EP). Like S-GMP it is a two-phase commit protocol that requires an initiator to block if it does not receive majority approval to its invitation. The majority requirement is necessary and sufficient to ensure that knowledge of the existence of any update that could have been committed will never be lost; the update’s existence, because it is process-functionality information, is propagated using a gossip scheme in responses to concurrently-issued and future invitations.

As in S-GMP, at the outset of execution, no EP membership service exists. Booting the EP membership service also involves a name service and a small, initial cadre of core members. Let Consensus\(^0\) denote this initial set.

The approval quota for local view updates is the number of processes from which approval to commit the update is required. As before, \(\text{Sizeof-Maj}(G)\) denotes the minimal size of a majority subset of \(G\); \(\text{Sizeof-Maj}(G) = \left\lfloor \frac{1}{2} G \right\rfloor + 1\). In EP the approval quota, \(AQ()\), for \(p\) is a function of \(\text{LocalView}_p(c)\), but in contrast to S-GMP there is an \textit{a priori} bound on the number of core members \(p\) can add to or remove from \(\text{LocalView}_p(c)\) in any single update. Since \(AQ(\text{LocalView}_p(c))\) must be at least a majority suppose:

\[
AQ(\text{LocalView}_p(c)) = \text{Sizeof-Maj}(\text{LocalView}_p(c)) + k
\]

where \(k\) is fixed, commonly agreed upon at the beginning of each run, and no more than the size of the largest minority subset of the initial consensus view \((0 \leq k \leq \left\lfloor \frac{\text{Consensus}^0}{2} \right\rfloor - 1)\). Then \(p\) can, in any single update instance, change
its view by at most $k + 1$ members. Intuitively, this restriction arises from the fact that the initial condition is the only commonly-known view upon which processes agree; it provides the only source of information from which processes can base an approval quota and still be assured that the set of approving processes will always intersect.

The choice of $k$ dictates how easily a process can alter its local view — larger values of $k$ require more coordination. In practical terms the initial consensus view will typically consist of no more than three processes, so it is likely that $k \leq 1$ in most cases, preventing processes from changing successive local views by more than two processes at any time throughout an entire run.

Finally, assuming the quota rule is a simple majority then providing Eventual Propagation requires at least $|\text{LocalView}_p(c)| + 2\text{SizeOfMaj}(|\text{LocalView}_p(c)|) - 3$ messages; $\text{EP}$ achieves this lower bound.

### 7.2.4 The $\text{EP}$ Algorithm

As in $\text{S-GMP}$ we do not explicitly show Disconnect, or error checking. Gossip, however, is more obvious: an outer member passes back to an initiator its local failure and recovery beliefs in the set $\text{ProcStatus}_p$. 
Task: Initiator, $init$

while ($\text{ProcStatus}_{init} \neq \emptyset$)

GetUpdate($EPvalue$, $\text{ProcStatus}_{init}$);

$\text{multicast}_{init}((\text{LocalView}_{init}), \text{EP-submit}(EPvalue))$;

for each $p \in \text{LocalView}_{init}()$

$\text{recv}_{init}(p, \text{ack}(\text{EP-submit}(EPvalue)) : \text{ProcStatus}_p)$ or $\text{faulty}_{init}(p)$;

if (majority didn’t respond) $\text{quit}_{init}$;

for each $p$ such that $\text{ProcStatus}_p \neq \emptyset$

$\text{ProcStatus}_{init} = \text{ProcStatus}_{init} \cup \text{ProcStatus}_p$;

$\text{DoCommit}(EPvalue)$;

$\text{multicast}_{init}((\text{LocalView}_{init}), \text{EP-commit}(EPvalue))$;

---

Task: Outer Processes, $p$

$\text{recv}_{p}(init, \text{EP-submit}(EPvalue))$;

$\text{send}_{p}(init, \text{ack}(\text{EP-submit}(EPvalue)) : \text{ProcStatus}_p)$;

$\text{recv}_{p}(init, \text{EP-commit}(EPvalue))$;

$\text{DoCommit}(EPvalue)$;

---

Figure 7.2: The Eventual Propagation Algorithm.
7.2.5 Strong GMP

The S-GMP protocol of Section 4.3 implements all four characteristic consistency properties. S-GMP uses the order it provides to rank processes and distinguish one, the \( \text{mgr} \), as responsible for initiating updates to the consensus view.

With S-GMP, one subtle cost of order is the restriction on process initiative: a lower-ranked process can initiate an update only when it believes higher-ranked processes faulty. Regarding message complexity, when \( \text{mgr} \) is believed faulty, the minimal cost to reconfigure depends on the degree of separation of local views and the degree of dissemination of the most recent update proposal. In only one failure scenario are three communication phases necessary incurring a cost of \( 2(\text{Sizeaf-Maj}(\text{LocalView}_p(c)) - 1) \) extra messages. In all other circumstances, S-GMP incurs no additional message cost over EP.

An intriguing aspect of S-GMP is that when changes to the consensus view are frequent, the cost of each update is less than that of either the VP or EP protocols. Order, Uniqueness, and Eventual Propagation combine to force any protocol implementing them to commit an update contingent upon the future removal of members currently believed faulty. The gossipy nature of our systems has the affect of making the 'contingency', a necessary part of the second-phase commit message, equivalent to a first-phase invitation message. As a result, when changes are frequent, the cost of a single update is amortized to \( O(2|\text{Consensus}_S(c)|) \) [RB91], cheaper than either of the other protocols.

Of course there are pathological failure scenarios in which S-GMP performs poorly. As in Section 5.3 when we counted messages, the worst case requires specific
processes to fail at the most inconvenient times: \textit{mgr} fails, then its replacement fails immediately before assuming control, then \textit{its} replacement fails immediately before assuming control, and so forth. However, in \textit{typical} system executions, while failures and recoveries may be frequent, that particular failure scenario is extremely improbable. In these situations \textsc{s-gmp} is, oddly, the least expensive of the three despite providing the strongest consistency properties.

In contrast to \textsc{ep}, the approval quota for \textsc{s-gmp} is always just a simple majority of a core member’s current local view. Most significantly, there is no bound on the number of processes that can be added\footnote{Obviously only a minority can be removed.} between consensus views! This is indeed unexpected as it allows successive consensus views to differ wildly. Again, the Order requirement lies at the heart of the explanation since it forces reconfigurers to query the outer members for their local states. It turns out that at every point in the execution where it might be possible for two update initiators to commit disparate views, both are competing for a majority subset of the same consensus view – only one of them can ‘win’, thereafter blocking the other and its approval cohorts.

### 7.3 Message Complexity Comparison

Tables 7.1 and 7.2 summarize Section 7.2. In \textsc{s-gmp}, for each \( p \in S \), \( \text{LocalView}_p(c) \) = \textbf{Consensus}_S(c). While \textsc{s-gmp} is more costly than \textsc{vp}, we believe the Virtual Partitions approach is ultimately more limited than either \textsc{ep} or \textsc{s-gmp} since it converges only in ‘quiet’ systems. \textsc{s-gmp} is no more costly than \textsc{ep} for all updates except \textit{mgr}’s removal from the core, and then only in a worst-case scenario. More-
Table 7.1: Minimal Message Cost for a Single Update to a Local View.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Consistency</th>
<th>Messages Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP</td>
<td>Conv wrt QUIET($S$)</td>
<td>$</td>
</tr>
<tr>
<td>EP</td>
<td>Limited Divergence</td>
<td>$</td>
</tr>
<tr>
<td>S-GMP: non-mgr</td>
<td>Order, Conv wrt True</td>
<td>$</td>
</tr>
<tr>
<td>S-GMP: mgr</td>
<td>No Divergence,</td>
<td>$</td>
</tr>
<tr>
<td>S-GMP: amortized</td>
<td>and Uniqueness</td>
<td>$\approx</td>
</tr>
</tbody>
</table>

Table 7.2: Internal Efficiency: Maximum Size Change to Local Views.

| Algorithm | $|Update|$ | Approval Quota |
|-----------|----------|----------------|
| VP        | add: $|\text{Proc}| - 1$ | $\mathcal{AQ}(\text{LocalView}_p(c)) = 1.$ |
| VP        | rmv: $|S| - 1$    | $\mathcal{AQ}(\text{LocalView}_p(c)) = \text{Sizeof-Maj}(\text{LocalView}_p(c)) + k$ |
|           | $\leq k + 1$ | where $0 \leq k \leq \left\lfloor \frac{|\text{Consensus}_S^0|}{2} \right\rfloor - 1$. |
|           |            | $k$ fixed throughout run. Usually $k \leq 1$. |
|           |            | Requires $|\text{LocalView}_p| > k$. |
| EP        | add: $|\text{Proc}| - 1$ | $\mathcal{AQ}(\text{LocalView}_p(c)) =$ |
| S-GMP     | rem: $\left\lfloor \frac{|\text{GpView}(c)|}{2} \right\rfloor - 1$ | $\mathcal{AQ}(\text{LocalView}_p(c)) =$ |
|           |            | $\text{Sizeof-Maj}(\text{LocalView}_p(c))$. |
over, we believe S-GMP will often be less expensive than EP since runs in which S-GMP can amortize its message cost (frequent joins and leaves) are far more probable than runs in which EP is cheaper (mgr and its replacements always failing at specific stages of the protocol).

This should seem contradictory: we are claiming that the more ordered protocol can be the least expensive and most productive to run. The intuition is that processes are using their knowledge of the strong ordering properties to pare down communication; the type and amount of information provided by the strong consistency guarantees allows processes to infer a great deal about the global environment independently – notably a consistent ranking of core members and the size difference and temporal distance between local views. A less powerful protocol cannot optimize in this fashion, resulting in a less-efficient scheme for performing the same updates. Thus, given an ordered protocol, a series of updates amortize message costs by exploiting knowledge of the strong ordering properties.

7.4 Using a Membership Service

The complexity results of the last section address only part of the cost-comparison issue that is the goal of this chapter. A second important question concerns the cost to applications of doing useful work over the various types of membership services.

To partially address this question, we consider the problem of managing a single-copy replicated data item. This is a classical problem encountered in both distributed and database systems; arguably, it is the most important problem actually solved in current real distributed systems.
Specifically, we consider algorithms permitting read and update access to a variable shared among the members of a process group. A correct solution should present the behavior of a single-copy, non-replicated variable. We do not consider database style transactions, despite the fact that transactions may amortize certain costs over a series of update operations and in this way reduce costs. Our reason for taking this approach reflects an interest in a wide range of distributed algorithms, including those cited in the introduction, which are typical of the functionality provided by a distributed programming environment such as the ISIS system. The cost of performing a single read or write to a shared variable is an accurate predictor of the cost of solving these sorts of problems. The cost of performing true database transactions, while important, is relevant primarily to a limited class of database-like applications.

For brevity, we assume the existence of a concurrency control mechanism ensuring that conflicting operations are never scheduled simultaneously (c.f. [ASC85]). In fact, synchronization mechanisms can be layered over any of the update algorithms described below; the points made below are of general applicability. Table 7.3 summarizes this section.

### 7.4.1 Virtual Partitions

Using the vp algorithm, reads can be done locally (or from any accessible copy of a variable), but every update to a shared variable requires a two-phase commit protocol. It is not hard to understand why this is the case. The membership consistency guarantees provided by the Virtual Partitions approach do not ensure
Table 7.3: Utility to Applications of MRM Guarantees

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>External Efficiency:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Work application must do to update replicated data</td>
</tr>
<tr>
<td>VP</td>
<td>2-phase commit required for EACH read and write.</td>
</tr>
<tr>
<td></td>
<td>Majority of data sites in updater’s virtual partition.</td>
</tr>
<tr>
<td>S-GMP</td>
<td>Reliable Multicast to Consensus$(c)$.</td>
</tr>
<tr>
<td></td>
<td>Flush only when Consensus$(c)$ changes.</td>
</tr>
</tbody>
</table>

that a process’s local view of its partition represents a possible global state in the actual system execution. In other words, a process can believe it belongs to a virtual partition that, in reality, never existed (as a consensus view among its partition cohorts) along any consistent cut in the execution. The first phase of the data update protocol essentially detects whether a process’s local view of its current partition is shared by the members of it; this is necessary because the one-copy serialization property is known to hold only if the virtual partition accurately reflects the communication structure at the time an update is done.

### 7.4.2 Eventual Propagation

Similarly the EP algorithm is an inadequate base for implementing a shared variable. Indeed, it is not at all clear that EP provides any meaningful information for managing replicated data. Recall that within this context, membership in a group should imply functionality and possession of a copy of a given variable. EP emits site failure and recovery information, but guarantees neither membership agreement
nor that the information is grounded in current reality: when \( \text{EP} \) tells \( p \) to add \( q \), it may be because \( p \) once belonged to some other process’s local view when that process added \( q \). Whether \( q \) is functional when \( p \) adds it is, under \( \text{EP} \), irrelevant. In this way a process’s local view provides no useful information.

As a result, it is unclear how to use \( \text{EP} \) in this context so that replicated updates can be ordered to ensure the one-copy property. Other than a quorum replication method [Her86], we know of no replicated data-management scheme that could be used to resolve this problem. Additionally, since \( \text{EP} \) provides no meaningful information, the cost of even this approach could be prohibitive: each read or write would involve interactions with multiple processes, and writes would again require some form of commit protocol. The result is that both read and write operations will be slow.

### 7.4.3 Strong GMP

The \( \text{s-GMP} \) algorithm, in comparison, offers an extremely lightweight environment for managing replicated data. In [Ste91], Stephenson showed how to implement a fault-tolerant, causally-consistent, totally-ordered multicast to group \( G \), using \( \text{s-GMP} \) as input, in at most \( 2 \times |G| \) (but usually \( |G| \)) messages. In [Jos86], Joseph showed that, for a large class of concurrency control mechanisms, a \( |G| \)-causal multicast suffices to perform a replicated update.\(^3\) As in \( \text{VP} \), reads are purely local, but by using such a multicast, writes no longer require a coordinated commit.

To understand why this is so, recall that in contrast to \( \text{VP} \) and \( \text{EP} \), \( \text{s-GMP} \) forces

\(^3\text{And to implement concurrency control if necessary.}\)
every local view to exist as a consensus view along some consistent cut in the execution. Though a process does not know precisely when that cut occurs, Order allows it to infer that the cut either exists ‘currently’ or will exist ‘soon’ – certainly before any other consensus view becomes defined. Consequently, every process executing an update is certain that the group upon which the update’s serializability depends is well-defined both in composition and with respect to its causal-temporal place in the execution. This assurance means that any data update can be committed safely, despite changes in the underlying group. As a result, the update protocol can be asynchronous – the initiator need not delay until the data reaches other group members.

To summarize, the S-GMP algorithm admits a cheaper and more asynchronous solution to the replicated data problem than either VP or EP. Of course this does not rule out the possibility of superimposing a layer on top of VP or EP that would effectively provide atomic multicast, but such a multicast would then achieve the properties of the S-GMP algorithm. The minimality proofs of Chapter 6 show that the resulting solution could never be less expensive than S-GMP.

Remarks

One might be curious about the practical importance of the distinction we made above; after all, the cost of running one or two extra rounds in a communication protocol will surely be small. However, experience with the ISIS system has been that similar results apply when solving any of a wide variety of distributed systems.

\footnote{We have not discussed failure-atomicity; this is done in [BSS91].}
problems using S-GMP as a system component. The predominant factor is the ability to issue operations asynchronously, especially beneficial in systems with bursty patterns of communication. Since operations can be emitted much faster than they can be sent they will, when issued asynchronously, queue up for transmission and can then be packaged, several to a message. This amortizes the physical cost of communication over a series of logical operations, making communication appear cheaper as the rate of operations rises. In contrast, multi-phase operation schemes are necessarily more synchronous since each successive operation involves independent communication; this required synchrony precludes the possibility of lessening the apparent cost of communication.

We conclude that the S-GMP protocol is more powerful in the properties it preserves, is generally (though not always) cheaper in terms of message complexity than the less ordered EP protocol, and is much cheaper than either VP or EP when used to solve a distributed data management problem.
Chapter 8

Implementation and Performance

In this chapter we discuss our MRM implementation. As mentioned in the Introduction, we view the MRM as a service whose clients can be any system process, but anticipating future plans, clients in these tests are all members of process groups. The MRM we implemented is a prototype for the Isis Toolkit’s new Failure Detector. The fundamental abstraction in Isis is the virtually synchronous process group [BJ87,Bir91]. Virtual synchrony masks distributed systems’ inherent asynchrony by reordering system events when necessary, so that communication appears synchronous and process omission and crash failures appear to be fail-stop [SS83, Sch84]. Specifically, the multicast protocols that implement virtual synchrony depend on group members observing group changes (members’ failures and recoveries) in the same order.
8.1 Measuring Performance

We designed the tests to pay specific attention to the physical system and its effect on MRM design, as well as how the MRM responds to stress in the environment. We measured two types of response time. *External response time* (ERT) is what MRM clients notice – the time between issuing a request and receiving the corresponding reply. Thus, ERT attempts to measure the cost our MRM inflicts on the overall system. Our current MRM, however, does more than simply order failures system-wide; it also takes care of client registration, failure detection, and event notification, each of which are logically distinct tasks and tax the MRM’s performance. While these tasks are necessary components of any MRM service, the MRM’s core members (i.e. those executing the S-GMP algorithm) need not perform them – that they do in this MRM implementation is one architectural choice. As a result, ERT may not be a completely accurate indicator of S-GMP performance. *Internal response time* (IRT) measures the time core members actually spend ordering failures and so quantifies S-GMP’s cost more directly efficiency. Internal response time is the difference between when a core member becomes aware of a request and when it commits the corresponding update. Figure 8.1 illustrates both kinds of latency.
Figure 8.1: External and Internal Response Time
8.2 Physical System

8.2.1 Network Layer and Machines

The communication layer is van Renesse’s Multicast Transport Service [vRBC+92], hereafter called MUTS. Communication in MUTS is between two types of entities – an *endpoint entity* is a port associated with a single listening process, while a *group entity* is a port with multiple listeners. MUTS provides reliable point-to-point and group-multicast communication and, since it runs on a variety of operating systems and transport protocols, a clean way to write portable, distributed programs. By allowing programmers to specify transport protocols, MUTS can be configured to take advantage of available system resources, or where necessary to adhere to specific system constraints. This portability is extremely desirable for our purposes as we wish the MRM to be as flexible and portable as possible.

MUTS also provides exactly the model assumptions we described in Chapter 2. It implements our communication assumptions (FIFO channels, point-to-point communication, and eventual, exactly-once delivery of uncorrupted messages), as well as a failure suspecter mechanism. The *monitoring subsystem* in MUTS (an artifact of the way MUTS implements eventual message delivery\(^1\)) alerts a message sender whenever a destination entity is unresponsive. Lastly MUTS approximates the countable process name space, \(\text{Proc}\), by assinging unique\(^2\) identifiers to new

---

\(^1\)MUTS retransmits a message until the destination acknowledges the message. While attempts to retransmit become less frequent over time, they do not cease unless the sender explicitly tells MUTS to give up. If the destination entity does not acknowledge the initial message, MUTS sets the destination entity's *suspicious* flag. If the entity sending the message had previously set a watch monitor on this destination, the suspicious flag triggers the monitor, alerting the sender that the destination is unresponsive.

\(^2\)Unique modulo a large number.
entities.

MRM core members ran on Sun Model 670 MPs. Clients executed on Sun SPARCstation1, and SPARCstation1+ workstations.

8.2.2 The Name Service and MRM Start-up

We have already mentioned the MRM’s dependence on a central name service. Our MRM implementation uses the MUTS name server, a directory mapping character strings to MUTS entity identifiers.

A process destined to be an MRM core member first looks up the MRM in the name server. If the MRM already exists, the process sends it a request to join the core; if the MRM is unknown to the name server, this process installs itself as the (initial core member of the) MRM. Because neither the name server nor the initial MRM are fault-tolerant we face a number of potential race conditions:

- ‘Multiple’ MRMs. Before any process succeeds in installing itself as the initial MRM, every process looking up the MRM concurrently, will attempt to install itself as the initial MRM. Because each maps its entity identifier to the same ASCII string (“MRM/mgr”) and because name server installations are atomic (i.e. indivisible), the first process to install itself as the initial MRM can receive and process requests until another of the initial processes installs itself. Each successive installation in the name server effectively overwrites all previous installations. While the previous processes will still believe they are the initial MRM, none will receive any future MRM queries after they have been overwritten. This situation can violate consistency guarantees since the
different initial MRMs may be handling MRM requests.

- No MRM: If the initial core member fails before the MRM reaches critical size\(^3\) then no MRM exists. Potential core members will get no response to their attempts to join the MRM, and potential clients will decide the MRM is unavailable.

A naive fix would have a potential core member time-out, retry (the MRM’s lack of response may be due to ongoing reconfiguring), then install itself as the new MRM. However, if the original MRM has not, in fact, experienced a total failure, the system will be partitioned with the original MRM and its clients operating on one side of the partition, and the ‘new’ MRM on the other. We are not yet sure how to solve this problem, or indeed whether it can be solved without relaxing the consistency guarantees and merging views once the partition disappears.

Similarly, if the name server fails the MRM is unavailable. We can make the name server more resilient by replicating it on different processors, but since it, like the MRM, must appear to be a single, fault-tolerant process, it must either be a client of the MRM or somehow employ S-GMP itself.

### 8.3 The Client-Server Interface

Client processes have two ports: a request port on which they make requests to the MRM and receive acknowledgements to these requests, and the event port on which

\(^3\)In our tests critical size was either one or two.
they receive failure and recovery event notifications from the MRM.

Clients must first register with the MRM. A client queries the system's name server to get the MRM registration site – in our case, mgr – which acknowledges the registration request. Because the registration site may have failed while a client is attempting to register, the client retries (querying the name server repeatedly) until it either decides the MRM is unavailable or receives an acknowledgement.

After registering, clients make requests to open groups, remove members from groups, close groups, and unregister themselves. We do not permit additions to groups; rather, a client opens a new group with the desired members. This puts a burden on clients to explicitly close groups that are no longer of interest, and clients lax in this regard can affect MRM performance. To help with garbage collection, a client can set a group's close-null (or, more generally, close-n) flag when it opens the group, indicating that the group should be closed when it is empty (or has fewer than n members).

Clients send their requests to the MRM request site, also mgr in this implementation, and receive formal acknowledgements to their requests. A client does not know that the MRM will definitely process a request until it gets an acknowledgement. The request site acknowledges a request only after distributing it to the entire MRM core. Clients keep unacknowledged requests in a buffer and reissue them upon being informed of a change in the request site.

A client receives event notifications from the MRM on its event port. An event notification informs a client that either:

- it is now an initial member of a new group, or
• it has been removed from a group, or

• it was a remaining member of a group that was closed, or

• the request site has changed.

Finally, once a client makes an unregisters it is no longer kept abreast of changes to the request site.

8.3.1 Requests and Event Notifications

Our MRM supports the following client requests:

• register(client-id): registers client-id as a client of the MRM. Hereafter, client-id can make further requests of the MRM, and will be informed whenever the request site changes.

• open(gp-id, entity-list, flags): create a group with the entities mentioned in entity-list as initial members. Duplicate entities are ignored. The entities in entity-list need not be registered clients of the MRM. Flags are:
  - new-id: asks the MRM to create a new gp-id through which all future reference to this group will be made;
  - close-null: close the group when it is null;
  - verify: actively check each member before removing it;
  - insulate: only honor the open() request if the client issuing the request is included in entity-list, and only honor future remove() requests from
current⁴ group members;

- kill-all: whenever a member of this group is removed from any other
group to which it belongs, also remove it from this group.

- query: forces the MRM to actively query the entities listed every \( t_1 \)
milliseconds. If the MRM does not receive responses within \( t_2 \) milliseconds,
it deems an entity faulty. \( t_1 \) and \( t_2 \) are MRM parameters.

- **remove**(gp-id, entity-list): remove the entities indicated from gp-id. The
  MRM interprets this as a hint that these entities are faulty. If gp-id’s insulate
  flag is set, the client issuing this request must be a member of gp-id as it exists
  at the time the request-site receives the request.

- **close**(gp-id): the group gp-id is no longer valid. If any members remain,
  remove them. If gp-id’s insulate flag is set, the same restrictions apply as in
  **remove();**

- **unregister**(client-id): indicates client-id is no longer interested in changes
  of the request-site. Remove all references to client-id from the MRM.

We assign responsibility for reporting event notifications by hashing the gp-id
to a core member’s rank. Clients receive the following event notifications from the
MRM:

- **MRM-open**(gp-id, entity-list): gp-id has been created with the entities men-
  tioned in entity-list as initial members. Only the entities listed receive this

---

⁴“Current” means, “At the time the request site received the request.”
notification. Each entity listed is distinct.

- **MRM-remove(gp-id, entity-list)**: the entities mentioned in entity-list have been removed from gp-id. Each entity listed was, in fact, a member of gp-id when the MRM executed the action. All remaining members of gp-id are sent this notification, as well as each member just removed;

- **MRM-close(gp-id)**: gp-id no longer exists at the MRM. This may be the result of removing members of gp-id so that the close-null flag required that gp-id be closed. If gp-id had any members remaining at the time it was closed, each receives the **MRM-close()** notification.

- **MRM-eventsite(new-event-site)**: new-event-site is the new MRM event site. Clients should re-send all unacknowledged requests to new-event-site.

### 8.4 Detecting Failures in the MRM

Our MRM currently detects failures in four ways:

- **remove(gp-id, entity-list)** requests from clients (Entity failures only). The MRM interprets **remove()** requests as hints that the listed entities are faulty. If gp-id’s verify flag is set, the MRM first queries each entity before deeming it faulty. If gp-id’s insulate flag is set, this hint is only accepted if the issuing client is a member of gp-id.

- **MUTS monitors** (Entity and core member failures). Core members set MUTS watch monitors on all other core members, all clients, and any non-client
entities of which it is aware. MUTS triggers a monitor whenever it is having trouble delivering a message – for example a request acknowledgement or an event notification.

- Active queries (Entity failures only). The MRM actively queries an entity if the group to which it belongs has its query flag set. The MRM deems faulty any entities not responding within a predetermined period of time. We did not use active queries in our tests.

- Time-outs (Core member failures only). A core member expecting a message from another core member waits between 10 and 30 ms (depending on overall system size, MRM size, and current frequency of client requests) before deeming the expected respondent faulty.

8.5 Testing the MRM

8.5.1 Parameters and Expectations

We studied MRM performance and scalability as a function of three parameters:

**Total System Size** is a function of the number of clients and entities and the number of groups to which they belong. The MRM maintains all system state in four hash tables:

- **Groups**: hashed by group entity identifier – maintains the current membership of each existing client group;
- **Opens**: hashed by group entity identifier – maintains the pending (i.e. uncommitted) open() requests;
- **Removes**: hashed by group entity identifier – maintains the pending `remove()` and `close()` requests;

- **Entities**: hashed by entity endpoint identifier – this data structure maintains all entities that are mentioned in the above three structures, cross-referenced by all groups to which the entity entity belongs.

As total system size grows these data structures become more expensive to access and update, since handling a request involves accessing both **Groups** and **Entities** once, and the appropriate update-pending hash table twice.

**Relevant-Event Frequency** refers to how often groups open and entities fail. Increasing relevant-event frequency stresses the MRM. We expect this to be evident in increased (perceived) MRM core failures, especially `mgr`, which as both the registration and the event site has a larger burden of MRM responsibility than other members. Response time will also grow as incoming requests will queue up while the MRM processes others.

**MRM Size** is the degree of replication. Larger MRM cores will be slower executing each update due to increased communication costs. On the other hand, the larger MRM will be more fault-tolerant making the failure of a single member less catastrophic. Also, in a larger core individual responsibility for event notification is reduced – each member spends more time doing event-ordering-related work, thereby increasing efficiency.

Less obvious system parameters affecting performance include:
• Initiator time-out interval: an update initiator (mgr or a reconfigurer) waits a pre-determined amount of time to receive outer members' acknowledgements before it deems them faulty. The shorter the interval the smaller the ERT, but the higher the risk of incorrectly deeming core members faulty. Such an error is costly: it decreases fault-tolerance appreciably, and adds to the remaining core members' responsibility (the departing member's reporting duties are distributed equally among the survivors), which can lead to further incorrect detections.

• Outer member time-out interval: an outer core member waits a pre-determined amount of time to receive a commit message from an initiator. If it does not receive the expected message in the time allotted, it deems the initiator faulty and, if warranted, initiates reconfiguration. Deeming an initiator faulty is especially expensive since our mgr serves as both request and registration site.

8.5.2 Testing Performance and Scalability

Simultaneously, on each of \( m \) machines, \( c \) clients concurrently executed a program in which each client further forked \( g \) concurrent threads. Each thread

1. issued an open() request to create a distinct group whose initial members were the \( c \) clients executing on that machine,

2. repeatedly

(a) slept for \( freq \) milliseconds,

(b) issued a remove() request to remove one member from the group,
until that client was the sole member, and finally

3. issued a `close()` request.

When all $g$ threads completed, each client wrote its external response data to file. Throughout a test, both clients and core members periodically recorded their host machine’s load.

We tested the MRM for a core size of one, and less extensively for a core size of three.\(^5\) MRM core members maintained a buffer to record their internal response data and wrote these to disk after every 1000 completed requests.

We present data for $m = 5$, for $c = 1$, $c = 5$, and $c = 10$, and for $g = 5$, $g = 10$ and $g = 20$.\(^6\) Finally, we show data for $freq = 2,000ms$, $freq = 500ms$, $freq = 50ms$, $freq = 5ms$, and $freq = 1ms$. Table 8.1 summarizes the results of all tests for the median ERTs. We ran three sets of tests (one for each of value of $c$), and averaged machine loads over a test’s duration. Table 8.2 summarizes the results of our tests for the median internal response times.

---

\(^5\)Since s-GMP requires majority agreement, an MRM with core size two is no more fault-tolerant than one with core size one.

\(^6\)We were limited in the number of groups per MUTS connection to 256.
Table 8.1: Median External Response Times (ms) - MRM of size 1

<table>
<thead>
<tr>
<th>freq = 2,000ms. open()</th>
<th>$c = 1$</th>
<th>$c = 5$</th>
<th>$c = 10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove()</td>
<td>[19,19,20]</td>
<td>[50,55,85]</td>
<td>[78,106,285]</td>
</tr>
<tr>
<td></td>
<td>[14,13,21]</td>
<td>[29,50,66]</td>
<td>[196,338,451]</td>
</tr>
<tr>
<td>freq = 500ms. open()</td>
<td>[18,18,18]</td>
<td>[42,68,138]</td>
<td>[125,166,393]</td>
</tr>
<tr>
<td>remove()</td>
<td>[12,12,12]</td>
<td>[42,80,168]</td>
<td>[221,235,403]</td>
</tr>
<tr>
<td>freq = 50ms. open()</td>
<td>[17,18,18]</td>
<td>[37,114,222]</td>
<td>[111,173,476]</td>
</tr>
<tr>
<td>remove()</td>
<td>[12,12,12]</td>
<td>[50,109,381]</td>
<td>[186,259,522]</td>
</tr>
<tr>
<td>freq = 5ms. open()</td>
<td>[18,18,17]</td>
<td>[53,144,253]</td>
<td>[209,221,445]</td>
</tr>
<tr>
<td>remove()</td>
<td>[12,12,12]</td>
<td>[53,208,388]</td>
<td>[316,305,450]</td>
</tr>
<tr>
<td>freq = 1ms. open()</td>
<td>[17,18,18]</td>
<td>[59,110,139]</td>
<td>[312,425,472]</td>
</tr>
<tr>
<td>remove()</td>
<td>[12,12,12]</td>
<td>[67,103,271]</td>
<td>[527,426,528]</td>
</tr>
<tr>
<td>Avg. Machine load</td>
<td>2.70</td>
<td>3.26</td>
<td>4.60</td>
</tr>
</tbody>
</table>
Table 8.2: Median Internal Response Times (ms) - MRM of size 1

<table>
<thead>
<tr>
<th></th>
<th>$c = 1$</th>
<th>$c = 5$</th>
<th>$c = 10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$freq = 2,000 ms. open()$</td>
<td>$g = [5, 10, 20]$</td>
<td>$g = [5, 10, 20]$</td>
<td>$g = [5, 10, 20]$</td>
</tr>
<tr>
<td>remove()</td>
<td>[3, 3, 3]</td>
<td>[4, 5, 4]</td>
<td>[4, 5, 5]</td>
</tr>
<tr>
<td></td>
<td>[3, 3, 3]</td>
<td>[5, 5, 5]</td>
<td>[5, 5, 7]</td>
</tr>
<tr>
<td>$freq = 500 ms. open()$</td>
<td>[3, 3, 3]</td>
<td>[4, 5, 5]</td>
<td>[4, 5, 5]</td>
</tr>
<tr>
<td>remove()</td>
<td>[3, 4, 3]</td>
<td>[4, 4, 5]</td>
<td>[4, 5, 5]</td>
</tr>
<tr>
<td>$freq = 50 ms. open()$</td>
<td>[3, 4, 4]</td>
<td>[3, 4, 4]</td>
<td>[5, 6, 6]</td>
</tr>
<tr>
<td>remove()</td>
<td>[3, 4, 4]</td>
<td>[4, 4, 5]</td>
<td>[5, 7, 7]</td>
</tr>
<tr>
<td>$freq = 5 ms. open()$</td>
<td>[4, 4, 4]</td>
<td>[4, 5, 4]</td>
<td>[4, 6, 4]</td>
</tr>
<tr>
<td>remove()</td>
<td>[4, 4, 4]</td>
<td>[4, 4, 4]</td>
<td>[7, 7, 7]</td>
</tr>
<tr>
<td>$freq = 1 ms. open()$</td>
<td>[4, 4, 4]</td>
<td>[4, 4, 5]</td>
<td>[5, 5, 5]</td>
</tr>
<tr>
<td>remove()</td>
<td>[4, 4, 4]</td>
<td>[4, 5, 7]</td>
<td>[8, 7, 8]</td>
</tr>
<tr>
<td>Avg. Machine load</td>
<td>0.59</td>
<td>1.09</td>
<td>1.72</td>
</tr>
</tbody>
</table>
Figure 8.2: Effect of Increasing Relevant-Event Frequency on External Response Time.
Figure 8.3: Effect of Increasing Relevant-Event Frequency on Internal Response Time.
Figure 8.4: Effect of Heavy-Weight Context Switch on External Response Time.
Figure 8.5: Effect of Light-Weight Context Switch on External Response Time.
8.6 Analysis

8.6.1 open() versus remove()

Our MRM algorithm is biased toward open() requests (mgr always chooses to process open() requests before remove() or close() requests\(^7\)) though this is not evident in the data. Interestingly, when the MRM is not taxed (c = 1 and some of c = 5) remove() requests are processed more quickly than open() requests because the protocol compacts independently-issued remove() requests that pertain to the same group.

Despite compacting, when there are many remove() requests issued at high frequency, they will remain in the Removes pending-update hash-table much longer. This is directly shown by the increased internal response time of remove() requests compared to open() requests, as total system size grows. The changes in ERT for both types of requests as system size grows are the result of other factors (e.g. context switching) we will discuss.

8.6.2 Increasing Relevant-Event Frequency

As expected, increasing the frequency of relevant events from one (per light-weight thread) every two seconds down to one every millisecond affects external and internal response time, except when c = 1. The MRM's request-site thread has higher priority than other MRM threads. Thus, when the time between events becomes shorter, the MRM receives requests at the expense of processing them, with the

\(^{7}\text{In fact, we must either bias toward open() requests to ensure that remove() requests are sensible: we cannot remove members from a group that has not yet been opened.}\)
result that requests stay in the pending-update tables longer. As above, this is shown more directly by the increase in internal response time as $freq$ diminishes when $c = 10$ (Figure 8.3). Interestingly, compacting seems to be significant in maintaining constant internal response time for $c = 5$.

### 8.6.3 Clients (Processes) Per Site

While increasing the number of clients per site significantly increases ERT (Figure 8.4), it does not have much effect on IRT. We attribute this to heavy-weight context-switching. The independent client processes executing on each site contend for processor time. Each client issues $g \times (c + 1)$ requests and receives as many formal acknowledgements from the MRM. A client need not, however, receive as many event notifications due to compacting. In fact, each client may receive as few as $2g$ notifications: one opening each group, and one closing each group, but even so, this makes $c(2g + 2g(c + 1))$ context switches per site.

With respect to MRM performance, external response time includes

1. the time for one MUTS rpc from the client to the MRM,

2. the time a request message spends (undelivered) at the request-site,

3. the internal response time,

4. the time for one MUTS rpc from the MRM to the client,

5. the time an event spends (undelivered) at the client’s event-notification port.

Other system factors are
1. heavy-weight context switch at client site,

2. light-weight context switch at client site, and

3. average machine load differences between clients and MRM.

In our tests, messages spend minimal time undelivered in their respective ports' queues – the MRM thread reading the request-site port and the client thread reading its event-notification port have highest priority. MUTS rpc is about $3ms[vRBC+92]$, and since the internal response time remains relatively constant for $c = 1$, $c = 5$, and $c = 10$ the heavy-weight context switch is the likely explanation.

### 8.6.4 Groups (Threads) Per Client

Similarly, increasing $g$ also increases external response time, though not as significantly as does increasing $c$ (Figure 8.5 reflecting the difference between light- and heavy-weight context switching.

### 8.6.5 Internal versus External Response Time

At first, the two orders of magnitude difference between internal and external response times may not be reassuring. However, the most significant factor seems to be context switching between heavy-weight client processes as well as between each processes' light-weight threads. Consider the case $c = 1$, $g = 5$, and allow $3ms$ for the client's request rpc to the MRM, and $3ms$ for the MRM's event-notification rpc to the client. Internal response times average $3.4ms$ (this includes the formal acknowledgement rpc), making $9.4ms$ we can account for. External response times for open() requests average $17.8ms$ (open() requests cannot be compacted) and
\texttt{remove()} requests average 12.4\textit{ms}, leaving 8.4\textit{ms} and 3.0\textit{ms} unaccounted. In terms of MRM performance, the 3.0\textit{ms} lower limit may be a combination of light-weight context switching and the time a message spends in an entity’s queue undelivered; it seems more likely that the 8.4\textit{ms} upper limit is the result of a context-switch to other, unrelated processes.

### 8.6.6 MRM Size

We performed some perfunctory tests varying the MRM core size. As Table 8.3 shows, clients surprisingly saw a decrease in external response time (with the exception of \textit{freq} = 50), perhaps attributable to core members now sharing responsibility for reporting event notifications. Presumably, the increased core size allows the MRM to execute more asynchronously. That ERT does not show that same characteristic rise as the number of groups per client increases supports this hypothesis. However, the tests are insufficient to conclude anything substantive.

### 8.7 Future Work

Certainly, we need to perform thorough tests on the replicated MRM core. At this time, the initial data imply that replicating the core improves ERT; we need to know at which point the cost of replication (communication within the MRM) surpasses this apparent benefit.

With respect to scalability, we can improve overall MRM performance by building distinct registration, detection, and reporting services. If the sole function of
### Table 8.3: Median External Response Times (ms) - $c = 5$

<table>
<thead>
<tr>
<th>freq = 500ms. open()</th>
<th>MRM size 3: $g = [5, 10, 20]$</th>
<th>MRM size 1: $g = [5, 10, 20]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove()</td>
<td>[50, 38, 34]</td>
<td>[42, 68, 138]</td>
</tr>
<tr>
<td>remove()</td>
<td>[27, 26, 27]</td>
<td>[42, 80, 168]</td>
</tr>
<tr>
<td>freq = 50ms. open()</td>
<td>[87, 149, 154]</td>
<td>[37, 114, 222]</td>
</tr>
<tr>
<td>remove()</td>
<td>[75, 205, 230]</td>
<td>[50, 109, 381]</td>
</tr>
<tr>
<td>freq = 5ms. open()</td>
<td>[51, 43, 25]</td>
<td>[53, 144, 253]</td>
</tr>
<tr>
<td>remove()</td>
<td>[67, 61, 58]</td>
<td>[53, 208, 388]</td>
</tr>
<tr>
<td>freq = 1ms. open()</td>
<td>[58, 49, 41]</td>
<td>[59, 110, 139]</td>
</tr>
<tr>
<td>remove()</td>
<td>[85, 44, 49]</td>
<td>[67, 103, 271]</td>
</tr>
<tr>
<td>Avg. Machine load</td>
<td>1.80</td>
<td>3.26</td>
</tr>
</tbody>
</table>
the MRM core is to order failures, these auxiliary functions represent inefficiency in the MRM, and probably detract from its overall performance.

Decoupling failure detection from the MRM is the area in which we believe we can realize the most significant improvements. We envision a set of filters that decide if a failure has occurred, leaving the MRM to simply decide when it occurred. By making the filters somewhat sophisticated we can also reduce the number of failures the MRM actually processes:

1. As the name implies, we could filter out duplicate suspicions, a decided advantage when a failed entity belongs to multiple groups since each group will generate a remove() request for the entity.

2. Similarly, by actively querying each entity before deeming it faulty, we could filter out many incorrect detections.

3. A filter could be quite discriminating. For example, it could ignore suspicions from processes that have a proven record of generating many inaccurate suspicions, and even deem such a process faulty.

4. Filters could also take better advantage of a wider variety of detection mechanisms. To keep the MRM as simple as possible, we limited the sources from which it obtains failure suspicions.

At present we try to minimize the time the MRM spends detecting failures. Except when a group’s verify flag is set, all suspicions are taken as ‘gospel truth’. As a result, we spend as little time as possible actually detecting failures and verifying a detection’s accuracy at the expense of processing more failure events.
Similarly, there is no need for any core member to handle client registration or event notification. In particular, because registration currently falls to a single process, that process’s responsibility to the MRM is much higher than other core members’, making the registration site not only a bottleneck, but also more likely to be deemed faulty. While responsibility for event notification should fall equally to all core members, event ordering and event reporting are nonetheless independent tasks. Core members would still be responsible for telling a Reporting Service about the event (for example, through the hashing scheme currently used), but would no longer be responsible for multicasting the event to a potentially large group. The ordering property is maintained outside the MRM since each event notification is stamped with the unique, sequentially-increasing MRM version its commit defined.
Chapter 9

Conclusions

9.1 Summary and Discussion

This thesis began with the belief that process membership is a vital system resource that must be managed consistently system-wide. We formally defined the class of process Group Membership Problems and examined one, Strong GMP, closely. We showed that Strong GMP is superior to other GMPs for specifying a Membership Resource Manager for distributed systems. In particular, we showed theoretically and experimentally that strong consistency guarantees and efficiency are not conflicting goals. This assertion has important ramifications for system design as well as for applications programming.

First, since we can ensure ordered, identical membership changes cheaply we can mimic the membership consistency of non-distributed systems by organizing a distributed environment around a central MRM. In this setting, system processes must rely on the MRM for all process status information. Preliminary experimental
results show that the system can be fairly large (in the number of processes as well as the size of the MRM’s load) without the MRM becoming a noticeable obstruction.

Second, membership consistency simplifies programming in distributed systems. Because distributed applications involve independent, asynchronous threads of execution, keeping track of and trying to account for all possible global states is a monumental task. Process membership information is arguably the most perplexing source of difficulty distributed applications programmers encounter. Any distributed application that requires processes to take some coordinated action (e.g. agree on a value, deliver messages in order, update a shared variable) requires some form of agreement on exactly which processes are taking the action. The difficulty arises because in asynchronous systems, crashed and slow processes are indistinguishable and also because processes' independent views of each others' functionality will differ. As a result the processes taking the action may each have different beliefs of their cohorts' identities. In this way, a central MRM precludes a major source of confusion.

9.2 Future Work

We would like to explore further the experimental work begun here. In particular, the implementation tested is inefficient and represents only a first pass at a realistic MRM. We wish to expand its portability to include more operating systems, and to take advantage of more sources of failure detections. In addition, we want to remove client registration, entity failure detection, and client reporting from the MRM; while these are necessary components of the total MRM service, they are
not integral to the main ordering role of the S-GMP protocol.

Once we have more appropriate scalability information, we will need to examine how multiple MRMs should interact in very large systems, as well as how they affect these systems. Similarly, to make the MRM service more flexible, we need to think about how to handle communication partitions. In our opinion, communication partitions are the single drawback to ensuring strong consistency. If the system partitions so that a majority of the MRM core is connected the service will be available to all clients in that partition, but to those in the minority, the MRM will be unavailable; in fact, the MRM will ‘kill’ the minority partition off. Alternatively, if the system partitions so that no majority subset of the core is connected, the MRM will not be available anywhere. One solution would allow the disparate MRM chunks to continue independently with the caveat that Strong GMP may be violated. We would then merge the chunks somehow when the partition is corrected, but certainly the issues involved in merging require further thought.
Bibliography


178


