IN-61-CR
14
p 35

Using Process Groups to Implement Failure Detection in Asynchronous Environments*

Aleta M. Ricciardi
Kenneth P. Birman

TR 91-1188
February 1991

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

*Research supported in part by DARPA/NASA Ames Grant NAG 2-593, and in part by grants from IBM and Siemens Corp.

Using Process Groups to Implement Failure Detection in Asynchronous Environments[1]

Aleta M. Ricciardi
Cornell University
Department of Computer Science
Ithaca, NY 14853 USA
aleta@cs.cornell.edu

Kenneth P. Birman
Cornell University
Department of Computer Science
Ithaca, NY 14853 USA
ken@cs.cornell.edu

February 7, 1991

[1]Research supported in part by DARPA/NASA Ames Grant NAG 2-593, and in part by grants from IBM and Siemens Corp.

Abstract

Agreement on the membership of a group of processes in a distributed system is a basic problem that arises in a wide range of applications. Such groups occur when a set of processes co-operate to perform some task, share memory, monitor one another, subdivide a computation, and so forth. In this paper we discuss the Group Membership Problem as it relates to failure detection in asynchronous, distributed systems. We present a rigorous, formal specification for group membership under this interpretation. We then present a solution for this problem that improves upon previous work.

Keywords Asynchronous computation; Fault detection; Fault tolerance; Distributed Consensus; Membership list management.

1 Introduction

Agreement on the membership of a group of processes in a distributed system is a basic problem that arises in a wide range of applications. Such groups occur when a set of processes co-operate to perform some task, share memory, monitor one another, subdivide a computation, and so forth. These problems are seen in data base contexts [2], real-time settings [7], and distributed control applications [14] [3]. A process group's membership may change when its processes *fail* (they are removed), *recover* (re-instated), when new processes join, or when members voluntarily leave. Some form of consensus on group membership is necessary, for without it a server that respects its specification may nonetheless behave inconsistently with some other server that has simply seen different group members. Cristian [8], specified and solved a similar problem for synchronous settings. This paper explores the problem in an *asynchronous* environment.

In our model of computation, a set of processes communicate through a completely connected network of reliable, FIFO channels. Processes only fail by crashing, and once failed, do not recover. We model process recovery by treating 'recovered' processes as new and different process instances. The system is fully asynchronous in that message delivery times are unbounded and there is no global clock.

Accurate detection of crashes (and recoveries) is impossible in an asynchronous environment. At best, a process can be *suspected* of having failed, but no process can ever be *known* to have crashed because real crashes are indistinguishable from communication delays. We therefore focus on what it means for a process to be a member of the group of *operational* processes in an asynchronous system. We model the presumed failure of a process by removing it from the group. The impossibility of detecting crashes also affects the meaning of *correct process*, for in the traditional literature a correct process is one that has not failed. In our setting, it is one that has not been *perceived* to have failed[1]. Our goal is to make this mechanism mimic a fail-stop failure detector.

Our approach and solution differ from previous work on group membership for asynchronous systems. In contrast to Moser, *et.al.* [16], we do not assume the existence of an underlying fault-tolerant atomic broadcast. Our solution is also cheaper than theirs, and the one proposed by Mishra, *et.al.* [15]. The protocol in Birman and Joseph [4] blocks during periods when failures and recoveries occur continuously. Our solution is fully 'online' : we can process a constant flow of requests to both remove and add processes, which is exactly what occurs in actual systems. Bruso's solution [5] is *symmetric* (*i.e.* all processes behave identically) and requires an order of magnitude more messages in all situations. Other, less directly related protocols include [6][2] and [9].

In Section 2 we discuss perceived failures in more detail and formally define the Group Membership Problem. We discuss why, despite its similarity to Distributed Consensus [10], GMP is solvable. In Sections 3 and 4 we construct our solution, initially considering failures but not recoveries. Section 5 proves the reconfiguration algorithm correct and provides crucial lemmas for correctness of the complete algorithm, which is shown in Section 6. We also discuss the protocol's complexity and minimality. In Section 7 we modify the original algorithm to allow processes to join the group. This yields a fully 'online' protocol in the sense that we can now process a constant flow of requests to both exclude and join processes, which is what occurs in actual systems. We conclude by discussing the implications of our particular specification, and directions for future work.

[1] If no other process attempts to interact with one that has, in fact, crashed, it will never be perceived to have failed

[2] Marzullo [13] has shown that the GMP protocol in [6] needs extension if failures or recoveries occur during the second phase of their "ring-reformation" protocol.

2 Perceived Failures

The notion of *perceived failure* is crucial to GMP in asynchronous environments. In this section we introduce our model and discuss perceived failures in detail. We also discuss how they affect GMP solutions and the meaning of *correct process.*

2.1 The System Model

Our system is of a set of n processes, Proc, communicating through complete set of reliable (lossless and non-generating), FIFO channels. There are no bounds on message transmission times, and no global clocks.

A *process history*, for a process $p \in$Proc, is a sequence of events including send events, receive events and specific internal events that arise in our algorithm. General internal computation is not modeled. We use $send(p, q, \mathrm{m})$ to denote the sending of message m by process p to process q, and $recv(p, q, \mathrm{m})$ to denote q's reception of m from p. A history for p is denoted

$$\mathrm{h}_p = start_p \cdot e_p^1 \cdot e_p^2 \cdots e_p^k, k \geq 0$$

where the e_p^i are events, and $start_p$ is a unique, internal event. Denote by $\mathrm{h}_p[i]$ the ith event of h_p, and by $|\mathrm{h}_p|$ the number of events of h_p.

A *system run* is an n-tuple of process histories, one for each $p_i \in$Proc. We say an event, e, is *in run* $r = (h_{p_1}, h_{p_2}, \ldots, h_{p_n})$, if e is an event in some process history, and denote this by $e \in r$.

Causality between events (denoted $\rightarrow$ and read *happens before*) in a given run is defined in the usual manner after Lamport [12]. In our model, a *consistent cut*, c, is a system run closed under $\rightarrow$; that is, if $e \rightarrow e'$ and $e' \in c$, then $e \in c$.

Definition Given two process histories, h_p and h_p', we say h_p is a *prefix* of h_p' if and only if $|\mathrm{h}_p| \leq |\mathrm{h}_p'|$ and $\forall 0 \leq i \leq |\mathrm{h}_p|.(\mathrm{h}_p[i] = \mathrm{h}_p'[i])$. h_p is a *strict prefix* of h_p' if and only if $|\mathrm{h}_p| < |\mathrm{h}_p'|$ and h_p is a prefix of h_p'.

Definition Given two consistent cuts, $c = (\mathrm{h}_1, \mathrm{h}_2, \ldots, \mathrm{h}_n)$ and $c' = (\mathrm{h}_1', \mathrm{h}_2', \ldots, \mathrm{h}_n')$, in the same system run,

1. $c \leq c'$ if and only if each h_p is a prefix of h_p';
2. $c \ll c'$ if and only if each h_p is a strict prefix of h_p'.

We model the crash failure of process p by a final event $quit_p$[3]. In this way, once p has crashed, it causally influences no other process. The proposition $down(p)$ holds along a consistent cut c exactly when $quit_p \in c$. We define $up(p) \equiv \neg down(p)$. Finally, we define $\mathcal{UP}(c) \subseteq$Proc, to be all processes p for which $up(p)$ holds along consistent cut c, and $\mathcal{DOWN}(c)$ to be Proc$-\mathcal{UP}(c)$. Asynchrony prevents processes from ever knowing the exact composition of $\mathcal{UP}(c)$ (except along the initial cut, $c_0 = (start_1, \ldots, start_n)$).

Perceived process failures may be triggered by a variety of phenomena. Complicating any algorithm for detecting failures is the possibility that a transient event could prevent a live process from sending or receiving messages, giving rise to spurious failure 'detections'. In such a situation one process might detect an apparent failure when another does not, simply by virtue of observing during a period of degraded performance. Any

[3] Whether a process actually executes this event is irrelevant (a process may be in an infinite loop); $quit_p$ is a convenience in modelling a process that permanently ceases communication with all others.

global characterization of a process as *operational* or *failed* will therefore require a distributed consensus protocol[4]. For brevity, we discuss only process failures, but there are analogous statements for 'recoveries'.

To model this, we treat failure *detections* as a form of input: the event $faulty_p(q)$ marks the point in p's execution when it decides that q is faulty. The proposition[5] $faulty_p(q)$ is true along a consistent cut, c, exactly when $faulty_p(q) \in c$. The possible sources of an event $faulty_p(q)$ are the following:

> **F1 :** (*Observation*) For whatever reason, process p determines that q has crashed. We are not concerned with the details of the mechanism used here, but for liveness, we do assume that it occurs in finite time after a real crash.

For example, p may be expecting a message from q and does not receive it within a pre-determined 'time-out' period (Note that we are using 'time' only as an (approximate) tool for detecting possible crash failures. Nowhere do we use time to reason about system state.).

> **F2 :** (*Gossip*) Process p receives a message m from some process, r, such that $faulty_r(q)$ — $send(r, p, \mathrm{m})$ and, when p executes $recv(r, p, \mathrm{m})$, it does not believe q faulty.

In both cases, p executes the event $faulty_p(q)$. Let $\Box, \Diamond, \boxminus$, and $\diamondminus$ (*henceforth, at some future point, always in the past,* and *some point in the past*) be tense logic modalities (See [18] for rigorous definitions). We also require :

> **S1 :** (*Isolation*) Once a processes, p, believes another, q, to be faulty, p never receives messages from q again.
>
> $$faulty_p(q) \Rightarrow \Box\neg(recv(q,p,\mathrm{m}))$$

Our protocol is such that some time after recording $faulty_p(q)$, p will execute the event $remove_p(q)$. We define the *membership view* for operational process p along cut $c = (\mathrm{h}_1, \mathrm{h}_2, \ldots, \mathrm{h}_n)$, (denoted $\mathsf{Memb}(p, c)$), to be the set p obtains by sequentially modifying its initial membership list according to the $remove_p(q)$ events in h_p. Trivially, we require $p \in \mathsf{Memb}(p, c)$. $\mathsf{Memb}(p, c)$ is undefined if $down(p)$ holds along c. Because h_p is linear, it makes sense to talk about the xth version of p's local view, which we denote Memb_p^x. The reader should notice that we distinguish between the events $faulty_p(q)$ and $remove_p(q)$. This is because we will require processes to coordinate updates to their local views. A process's initial, local decision about another's faultiness must be propagated to all cohorts before removal can take place.

We extend local views to *system views* as follows.

Definition Given a consistent cut c and a set of processes, $S \subseteq \mathsf{Proc}$:

$$\mathsf{Sys}(c, S) = \begin{cases} \emptyset & S \cap \mathcal{UP}(c) = \emptyset \\ \mathsf{Memb}(p, c) & \forall p, q \in S \cap \mathcal{UP}(c).(\mathsf{Memb}(p, c) = \mathsf{Memb}(q, c)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We say that S is the set of processes that *determine* the system view.

[4] This is *not* Distributed Consensus as defined in [10].

[5] In general, we write events in *italics*, and propositions in *slanted* type.

2.2 Relating $\mathsf{Sys}(c, S)$ to Failure Detection

As the definition of $\mathsf{Sys}(c, S)$ is crucial to our Group Membership Problem, it is worthwhile discussing some subtle points. Intuitively, $\mathsf{Sys}(c, S)$ models the set of processes that the members of S believe operational along consist cut c. During periods of quiescence, we will want $\mathsf{Sys}(c, S) = S$. During periods of activity, we will be particularly interested in how the sets S and $\mathsf{Sys}(c, S)$ relate.

Assume $\mathsf{Sys}(c, S)$ is defined and suppose q is not a member of the group whose local views determine the system view; *i.e.*,$q \in \overline{S}$. Then $\mathsf{Memb}(q, c)$ need not be identical to other processes' local views for the system view to exist. Our concern lies with q taking an external action that reflects an incorrect composition of the system view. If q is truly failed this is impossible. So consider $q \in (\overline{S} \cap \mathcal{UP}(c))$. Two cases are of particular interest.

1. $q \in (\overline{S} \cap \mathcal{UP}(c) \cap \overline{\mathsf{Sys}(c, S)})$. In words, q is functioning, but is a member of neither the system view nor the group determining the system view. Given our intuition that $\mathsf{Sys}(c, S)$ is the set of processes mutually believing each other to be operational, communication should remain within this group. However, as q is operational along c, it may try to send messages to those in $\mathsf{Sys}(c, S)$. To effect our intuition, via system property S1, we would like a rule of the form $q \notin \mathsf{Memb}(p, c) \Rightarrow faulty_p(q)$, which would inhibit p from receiving messages from any q not in its local view. This prevents a process not in the system view from influencing those in it.

2. $q \in (\overline{S} \cap \mathcal{UP}(c) \cap \mathsf{Sys}(c, S))$. In words, q is a functional member of the system view but not a member of the group determining the system view. As $q \notin S$, its local view may contradict the system view, and this, given our interpretation of $\mathsf{Sys}(c, S)$, represents an *inconsistency* in the system state. Our goal is to avoid the danger of this occurring. We would, therefore, like a rule requiring q to be in S whenever it is in $\mathsf{Sys}(c, S)$; $(\mathsf{Sys}(c, S) \cap \mathcal{UP}(c)) \subseteq (S \cap \mathcal{UP}(c))$. It is easy to see that $(S \cap \mathcal{UP}(c)) \subseteq (\mathsf{Sys}(c, S) \cap \mathcal{UP}(c))$. Thus, we will require $S = \mathsf{Sys}(c, S)$.

2.3 Problem Description : The Group Membership Problem

We proceed to a formal definition of the Group Membership Problem (GMP) as it relates to failure detection in asynchronous systems. This consists of defining a safe and live distributed algorithm whereby processes may query $\mathsf{Memb}(p, c)$ during execution, and such that operational processes observe "1-copy" behavior on the sequence of views so-obtained (*i.e.* all see the same sequence of view transitions). Because responses to queries on $\mathsf{Memb}(p, c)$ will be taken as reflecting an exact system view composition, we will want to ensure that processes see *identical* sequences of view transitions. Failed processes will see only a prefix of all view transitions, but their local views when they are operational must not be permitted to diverge.

Since, in our model, logical formulas are true along consistent cuts, we omit explicit reference to particular cuts in the formulas. For example, the logical formula $q \in \mathsf{Memb}(p)$ is true along only those consistent cuts c for which $q \in \mathsf{Memb}(p, c)$. We define the proposition $out(p)$ to hold along all consistent cuts c for which $p \notin \mathsf{Sys}(c, S)$, and $in(p)$ to hold when $p \in \mathsf{Sys}(c, S)$.

An algorithm solves asynchronous GMP if each of the following properties are satisfied :

GMP-0 The initial system view, Sys^0 , exists along the initial cut; $\mathsf{Proc}=\mathsf{Sys}(c_0, \mathsf{Proc})$.

GMP-1 A process does not remove another process from its local view capriciously;

$q \notin \mathsf{Memb}(p) \Rightarrow faulty_p(q)$.

GMP-2 In every system run there exists a unique (denoted $\exists\,!$) sequence of system views upon which the functional members of each view agree;

$$\forall r.\ \exists\,!\ \mathsf{Views}(r) = \{(c_0, S_0), \ldots, (c_k, S_k) \mid (0 \leq k) \wedge (c_x \ll c_{x+1}) \wedge \textsc{Sys}(c_x, S_x) = S_x\}$$

Because the cuts are non-intersecting and unique, it makes sense to talk about the xth version of the system view, which we denote Sys^x .

GMP-3 All processes see the same sequence of local views, provided the views are defined; $\forall p, q.(\forall 0 \leq x.(\mathsf{Memb}_p^x = \mathsf{Memb}_q^x))$. This is equivalent to requiring each local view to eventually become a system view.

GMP-4 Processes are never re-instated to local views; $q \notin \mathsf{Memb}(p) \Rightarrow \Box(q \notin \mathsf{Memb}(p))$

GMP-5 For each event $faulty_p(q)$, and $p \in \mathsf{Sys}^x$, eventually either p or q is removed from the system view; $faulty_p(q) \Rightarrow (\Diamond(out(q)) \vee \Diamond(out(p)))$.

A few points are of note here. First, because our detection mechanism operates in finite time, a crash failure will be detected by any process dependent upon the failed one. GMP-1 and property S1 isolate faulty processes.

Second, notice that 'failure detections' by 'faulty' processes are finessed by these conditions. On the one hand, property GMP-5 forces processes, and therefore the system, to react to failure detections. This, also rules out the trivial solution. On the other hand, S1 causes messages from suspected faulty processes to be ignored (actually discarded), implying that if a process p makes a detection $faulty_p(q)$ and some other process concurrently believes p to have failed, it may be that no operational process will learn of p's detection. If the detection was erroneous and q is operational, the event $faulty_p(q)$ may or may not trigger q's eventual exclusion. The outcome will depend on the pattern of communication that ensues.

Finally, observe that, as an artifact of GMP-1, there is an implied composition of the various system views. Specifically, given c_x, and $q \in \mathsf{Sys}^{x-1}$, if for no process $p \in \mathsf{Sys}^{x-1}$ does $faulty_p(q)$ hold along c_x then q must be in Sys^x . Thus, we have captured the intuitive notion that system views represent processes that are mutually believed operational.

2.4 Difference Between GMP and Distributed Consensus

Our safety and liveness properties both define GMP as well as distinguish it from Distributed Consensus (DC) [10]. Though it appears very similar to GMP, DC is strictly stronger.

In DC, at least one process must reach a decision on a bit value. This decision is final. Moreover, all processes reaching decisions in a given run must choose the *same* value. Finally, both outcomes are possible, ruling out the trivial solution.

Since processes are required to reach the same decision, once a process reaches a decision, all other processes must eventually have knowledge of that decision value (else they could decide the other). This is

exactly Halpern and Moses's *eventual knowledge* [11], and, by the induction rules for eventual knowledge, DC would attain *Eventual Common Knowledge*. Halpern and Moses prove that, when communication is not guaranteed, Eventual Common Knowledge, and therefore DC, can not be attained.

GMP, on the other hand, is phrased in terms of Concurrent Knowledge, which is knowledge achieved along a consistent cut [17]. Concurrent Knowledge is weaker that Eventual Knowledge, but, we believe, appropriate for asynchronous systems. Moreover, GMP is not required to attain concurrent common knowledge[6]. The Appendix of this paper contains a detailed epistemic analysis of GMP. Finally, GMP uses a modified notion of correct process, allowing us to discount some processes that may not, in reality, be crashed. For these reasons, the impossibility result does not apply to our work.

3 Solution

Our solution to GMP will make use of two channel properties, one of which is not immediate from the model. First, we require channels to be FIFO, and second, we require that there be no messages from future views. Both of these properties are easily implemented : the former requires a (1-bit) sequence number on each message and an acknowledgement protocol; the latter involves adding view numbers to messages so that they can be delayed when received from a process in a future view (*i.e.* until that view is installed locally).

3.1 The Basic Algorithm - Mgr Does Not Fail

Our solution to the Group Membership Problem is asymmetric : it involves a distinct process, denoted Mgr, responsible for coordinating updates to the outer processes' local views. We use a *two-phase* protocol when Mgr co-ordinates local updates, and a *three-phase* protocol to select a new co-ordinator and stabilize the system when Mgr is perceived to have failed. To introduce the structure, we initially assume Mgr does not fail. We also modify the local views only by deleting process identifiers from it. Mgr 's failure and join operations are considered later.

In accordance with GMP-5, when a process p executes the event $faulty_p(q)$, it sends a message to Mgr, requesting that it start the removal algorithm. Every process, upon noting $faulty_*(q)$, disconnects its incoming communication channel from q, thereby satisfying S1.

Mgr initiates the two-phase update algorithm when it becomes aware of a failure. In Phase I (Figure 1) Mgr broadcasts a removal *invitation* message, Exclude(q), and awaits the outer processes' responses or notification of their failure. In this way, at the end of Phase I, all non-faulty processes (from Mgr 's perspective) believe *faulty*(q). In Phase II, Mgr broadcasts a removal *commit* message, Commit(q), telling processes that (weak) consensus on q's failure has been reached and they can remove q from their local views. Processes Mgr believes faulty will not participate in the update algorithms. Thus, the agreement on a new system view becomes *contingent* upon the subsequent removal of these processes. System property F2 ensures that operational outer processes become aware of such contingencies. Because Mgr is a single process, the outer processes' local views at the end of each invocation of the two-phase algorithm are identical.

Observe that the invitation message, Exclude(q), is unnecessary (with respect to GMP-1) if Mgr knows the outer processes already believe q faulty. In this way, the contingent updates, piggy-backed upon a commit message, serve as an invitation for subsequent view changes. We can thus compress successive rounds if

[6] GMP need not even attain *eventual* concurrent common knowledge, defined analogously to $C^{\diamond}$.

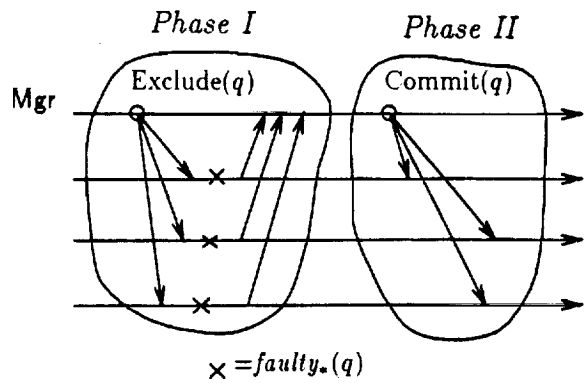

Figure 1: Structure of the Two-Phase Protocol

Mgr makes known, when it issues the Phase II broadcast for the current change, how it plans to change the system view next.

Conventions and Notation

We use different type styles in writing different objects. Events are written in *emphasized* type ($quit_p$, $faulty_p(q)$), and variables are in sans serif type (Mgr, Faulty(p), Ch_p). Program key words are in **bold face** type (**begin**, **await**), and formulas of the logic are in *slanted* type ($up(p)$, $faulty_p(q)$).

We will also adopt conventions in the figures that follow. Process histories are represented by horizontal rays. A solid (diagonal) ray between two process histories represents a message from the ray's source to its sink. Dashed rays indicate messages whose existence is hypothetical, in the sense that no direct information is available to indicate whether this message was sent. A solid line emanating from one process history and terminating without reaching another history represents a message that cannot be received due to system property S1. A set of messages grouped at the sender with an open circle represents a broadcast, as defined below. Message contents, when necessary, will be indicated in text near the ray's source. We will also indicate particular events or points of note in processes' executions as needed.

Let Memb_p^0=Proc, for all p and all runs. We use Memb(p) to denote p's current local view, when a cut is clear from context.

Given *send*(p, q,m), let *Bcast*(p,G,m) be the action $\forall q \in$G.(*send*(p, q,m)) where G is a set of processes. *Bcast*(p,G,m) is an indivisible action in the sense that p does not execute any other events until all messages are sent, but it is not failure-atomic.

Faulty(p) is a set of processes, local to p, which p believes faulty but has not yet removed from Memb(p). Ch_p is the set of channel id's connected to process p. The channel (p, q) is in the direction *from p to q*.

At startup, we assume the initial group membership, Proc, is commonly-known. We also assume that the event $faulty_p(q)$ triggers the appropriate actions regarding Faulty(p), as well as disconnecting the incoming channel (p, q).

```
Mgr
    Begin :
    while true do
        begin
        await (Faulty(Mgr) ≠ ∅);
        proc-id ← delete (Faulty(Mgr));
        while (proc-id≠ nil-id) do
            begin
            Bcast(Mgr,Memb(Mgr),Exclude(proc-id));
X.1         ∀p ∈ Memb(Mgr).(await (OK(p) or faulty_Mgr(p)));
            remove_Mgr(proc-id);
            GetNext(next-id);
            Bcast(Mgr,Memb(Mgr),Commit(proc-id): Contingent(next-id:Faulty(Mgr)));
            proc-id ← next-id;
            end ;
        end ;
    End.
Outer Processes
    Begin :
    recv(Mgr, p,Exclude(proc-id))
    if p = proc-id then quit_p.
    faulty_p(proc-id);
X.2 repeat send(p, Mgr,OK(p))
        await (Commit(proc-id):Contingent(next-id:L));
        if (p ∈ L) or (p =next-id) then quit_p.
        faulty_p(next-id);
        ∀l ∈ L.(faulty_p(l));
        [note: faulty_p(nil-id) is a null operation]
        remove_p(proc-id);
        Faulty(p) ← Faulty(p) − {proc-id};
        proc-id ← next-id;
    until (proc-id=nil-id);
End.
```

Figure 2: Exclusion Algorithm

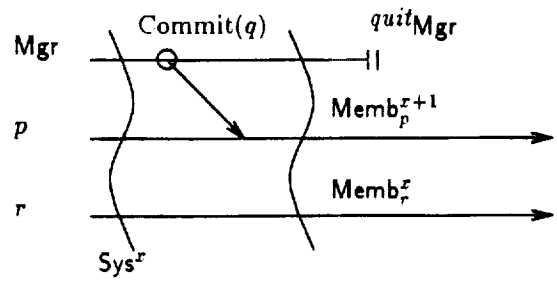

Figure 3: Inconsistent System

Remarks

This protocol can tolerate $|\mathsf{Memb}(\mathsf{Mgr})|-1$ failures. We will see that fault-tolerance decreases appreciably when **Mgr** can fail; only a minority of failures can be tolerated between successive system views.

4 Reconfiguring - Allowing Mgr to Fail

In this section we present a *reconfiguration algorithm* that selects a new co-ordinator (new **Mgr**) and stabilizes the system when **Mgr** is perceived to fail.

If **Mgr** fails in the middle of an update commit broadcast no system view will exist (see Figure 3). To re-establish the system view, our reconfiguration algorithm must address two problems : *succession* - which process(es) should initiate the reconfiguration algorithm and which should assume the **Mgr** role at the end; *progression* - which system view should a reconfiguration initiator propose to resolve inconsistencies and maintain safety.

Intuitively, reconfiguration depends on an initiator's ability to determine the last defined system view and propagate the correct proposal for the succeeding system view. In our algorithm, all successful reconfigurers (those able to reach the commit phase), undertaking reconfiguration of the xth system view, determine identical proposals.

GMP-2 requires system views to be unique. This forces any initiator to obtain responses from a *majority* of processes in its local view. An initiator can fail to obtain a majority in three ways : the initiator, itself, may be faulty, the network may be partitioned, or a majority of processes may be faulty. In the last instance, no algorithm can make progress unless some recoveries occur.

GMP-3 forces us to account for *invisible commits*. These occur when the only processes receiving a commit message fail. While no subsequent reconfiguration initiator will ever *know* whether any commit messages were sent, if an invisible commit did occur, the system must behave in a manner consistent with that event. This is the most difficult aspect of reconfiguration, as it is imperative that every invisibly committed update be detectable by every successful reconfigurer. We can ensure this only if the degree of system-wide inconsistency is tightly-enough bounded so that any initiator obtaining a majority of responses in the interrogation phase can infer the composition of local views of processes not responding to it. That is, local views must not be permitted to diverge so far that majority subsets might not intersect[7].

[7] This is also relevant to GMP-2, for ensuring unique system views requires at most one initiator to be able to obtain responses from a majority of processes in its *local* view.

In our algorithm, all *successful* reconfigurers attempting to install[8] the xth system view propagate Mgr 's proposal, if they become aware of it, and if not, propose Mgr 's removal. Unfortunately, asynchrony and inopportune failures can result in there being two different proposals for the same instance of the system view. We prove only one of them could possibly have reached the commit stage (we call such a proposal *stably-defined*), and then that any reconfigurer can determine which one it is. By propagating the stably-defined proposal, a reconfigurer forces the system to act consistently with any possible invisible commits. Moreover, we ensure that all stably-defined proposals for the same version number are identical, further ensuring GMP-3 as no process commits a local view for version x that differs from another process's version x.

4.1 Structure of the Reconfiguration Algorithm

Unlike the exclusion algorithm, the reconfiguration algorithm requires three phases. This is interesting and important, though not surprising in light of Skeen's work on non-blocking commit protocols [20]. In the first phase, the initiator broadcasts a reconfiguration *interrogation* message to all processes in its local view and awaits their responses[9,10]. If a majority respond, the initiator determines an update event, based on the outer processes' local states, whose execution would restore the system view. The initiator broadcasts this event as the reconfiguration *proposal* message[11]. After receiving another majority response, the initiator broadcasts a reconfiguration *commit* message. Majority responses are essential in maintaining GMP-2 and GMP-3; without it, the initiator must block.

4.2 Rules of Succession

We solve the succession problem by assuming a deterministic, *linear ranking* on process identifiers, with Mgr the highest-ranked process [12]. We say p has *higher rank* than q if $\mathsf{rank}(p) > \mathsf{rank}(q)$. Whenever a process is removed from a view, the ranks of all lower-ranked processes are increased by one. The rank of an excluded process is undefined. Thus, in the xth system view, $\mathsf{rank}(\mathsf{Mgr}) = |\mathsf{Sys}^x|$, and $\mathsf{rank}(p) = 1$ if p is lowest-ranked process. Observe that while p and q are in the same system views, 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 $\mathsf{Memb}(p, c)$

$$initiate(p) \equiv \bigwedge_{q \in \mathsf{Memb}(p,c)} ((\mathsf{rank}(q) > \mathsf{rank}(p)) \wedge faulty_p(q))$$

While this could lead to multiple, concurrent reconfiguration initiations, it guarantees that at least one process will begin the reconfiguration algorithm. Consider Table 1 in which $\mathsf{rank}(\mathsf{Mgr}) = x$, $\mathsf{rank}(p) = x - 1$, and $\mathsf{rank}(q) = x - 2$, and both p and q believe Mgr to be faulty. In the third scenario, both processes initiate reconfigurations. Section 4.3 discusses how multiple, concurrent reconfiguration attempts could affect view

[8] or complete the installation of

[9] More precisely, it awaits their responses or executes *faulty*().

[10] Observe that it will be necessary to over-ride the message buffering mechanism to be able to reconfigure from a version-inconsistent state. We therefore assume that neither interrogation nor responses nor commit messages will be buffered.

[11] The proposal may be a sequence of events. Its size is a function of the current size of the system view and must guarantee that majority subsets of $\mathsf{Memb}(r)$ and $\mathsf{Memb}(r) - \{Proposals\}$ intersect. Section 5 explains this necessity in more detail.

[12] Process rank is, in fact, based on 'seniority' with respect to duration in the system view

p actual state	q thinks p	q initiates?	p initiates?
Up	Up	No	Yes
Failed	Up	Eventually	No
Up	Failed	Yes	Yes
Failed	Failed	Yes	No

Table 1: Multiple Reconfiguration Initiations

uniqueness. In the second scenario, q expects p to initiate a reconfiguration. Eventually, q will "time-out" on p, surmising $faulty_q(p)$, and initiate the reconfiguration.

To implement the initiation rule, each process, p, maintains a local list, HiFaulty(p) with maximum size rank(p)-1, whose contents are the id-s of all higher-ranked processes, still members of Memb(p), that p believes faulty. Processes in HiFaulty(p) are removed from it upon their removal from Memb(p), and HiFaulty(p)'s maximum size is decreased by one.

HiFaulty(r) : A set local to process r, of size |Memb(r)|$-$rank(r), updated as follows :

1. Upon noting $faulty_r(q)$ for q of *higher* rank than r, q is added to HiFaulty(r).
2. Upon removing q from Memb(r), q is removed from HiFaulty(r). The size of HiFaulty(r) is decreased by 1.

4.3 Maintaining GMP-2 : Uniqueness of the Reconfiguration View

Recall that GMP-2 requires that the system view installed by reconfiguration (and removal) be unique. Consider the following situation, depicted in Figure 4, in which Q and R are subsets of Proc:

1. $faulty_Q(r)$ $\rightarrow$ $recv(q, Q,$Interrogate) $\rightarrow$ $recv(q, Q,$Commit(RL$_q$):Contingent(Faulty(q))) for each process q' in Q
2. $faulty_R(q)$ $\rightarrow$ $recv(r, R,$Interrogate) $\rightarrow$ $recv(r, R,$Commit(RL$_r$):Contingent(Faulty(r))) for each process r' in R.

Inconsistency may arise since no process in Q will receive r's interrogation (Figure 4) and no process in R will receive q's. Uniqueness of the system view would eventually be violated.

To prevent this and ensure that only one process (at a time) succeeds in installing a reconfiguration view, we require any initiator to obtain responses from a *majority* of processes in its local view. Let

- $\mu_{r,c} = \lfloor \frac{|\mathsf{Memb}(r,c)|}{2} \rfloor + 1$. We will write μ_r, when c is understood.
- $\mu_r^x = \lfloor \frac{|\mathsf{Memb}_r^x|}{2} \rfloor + 1; \mu^x = \lfloor \frac{|\mathsf{Sys}^x|}{2} \rfloor + 1$.

Let Phase1Resp(r), for reconfiguration initiator$r$, be r together with the processes responding to its interrogation, and Phase2Resp(r) be r and the processes responding to its proposal. Then r, beginning in local view Memb$_r^x$, can succeed in proposing a reconfiguration system view if and only if |Phase1Resp(r)|$\geq \mu_r^x$, and can succeed in committing[13] the proposed view if and only if |Phase2Resp(r)|$\geq \mu_r^x$. An initiator that is

[13] To completely ensure that only one process succeeds in installing a view, we must also bound the size difference between two processes' local views; if not, then majorities need not intersect. We discuss this in more detail.

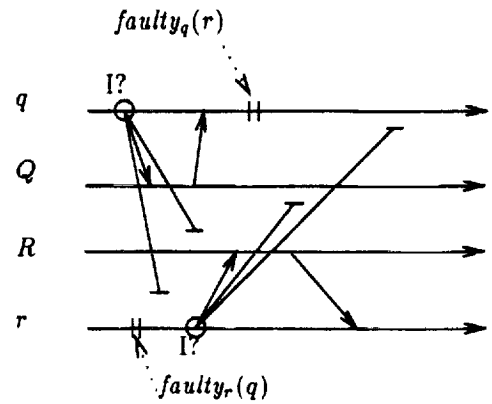

Figure 4: Majority of Responses Needed

unable to obtain either majority will execute *quit*. An initiator can fail to obtain a majority in three ways : the initiator, itself, may be faulty, the network may be partitioned, or a majority of processes may be faulty. In the last instance no algorithm can make progress unless some recoveries occur.

4.4 Maintaining GMP-3 : Propagating Committed Local Views

The content of outer processes' Phase I responses should allow the initiator to determine the nature and composition of any version-inconsistency. While local view information suffices to detect inconsistencies between the processes responding to an initiator, it falls short of satisfying GMP-3 entirely as invisible commits are not detectable.

To communicate its local view, a process responds with the sequence of *remove*() events it has executed, which we denote by seq(p) for process p. To aid in detecting invisible commits, each process maintains a list of triples, next(p), indicating how it expects to change its local view next. For example, the triple $(-p_r : r : x)$ means p is expecting a commit message from r, ordering p_r's removal from Memb(p), and resulting in the xth system view. Let ver(p)= x hold along all consistent cuts, c, for which Memb(p, c)=Memb$_p^x$ and define next(p) as follows :

- next(p) $\leftarrow$ ($-q$:Mgr:x) once p responds to *recv*(Mgr, p,Invite(q)) and ver(Mgr)=ver(p)= $x-1$.
- next(p) $\leftarrow$ ($-$next-id:Mgr:x) once p responds to a removal commit message Commit((proc-id):(next-id:Faulty(Mgr))) and ver(Mgr) = ver(p)= $x-1$. If next-id is the nil-id, next(p) is simply ($\emptyset$: Mgr : x).
- let $\mathcal{N}$ = next(p). Then next(p) $\leftarrow$ ($\mathcal{N}$,(? : r :?)), once p responds to *recv*(r, p,Interrogate).
- let ($\mathcal{N}$,(?:r :?)) = next(p). Then next(p) $\leftarrow$ ($-$RL$_r$: r : x) once p responds to *recv*(r, p,Propose(RL$_r$:x)). It is not hard to see that when p receives r's proposal message, the last element of next(p) must be (? : r :?).
- Otherwise, p is not awaiting any commit or proposal message, and next(p) is empty.

When ver(p)= x, the succession rule and S1 mean next(p) is the proposal of the *lowest*-ranked among all processes from which p receives proposals for version $x+1$.

Given a proposal, $\pi = (p_r : r : x)$, of $\mathsf{next}(p)$ we use $1st(\pi)$ to obtain p_r, $2nd(\pi)$ to obtain r, and $3rd(\pi)$ to obtain x.

Definition Let r be a reconfiguration initiator. Then $\pi = (z : p : v)$ is *committed invisibly to* r if and only if $\exists q' \notin \mathsf{Phase1Resp}(r).(z \in \mathsf{seq}(q')) \wedge \forall q \in \mathsf{Phase1Resp}(r).(z \notin \mathsf{seq}(q))$.

4.5 The Reconfiguration Algorithm

Note that while $\mathsf{HiFaulty}(p)$ is local to each process p, rank is commonly known. Consequently, other processes can infer the contents of $\mathsf{HiFaulty}(p)$ in the event that p initiates a reconfiguration. The variable 'invis' refers to the first process r will remove after successfully reconfiguring the system. $\mathsf{ProposalsForVer}(x,r)$ is a set, local to reconfigurer r, of the processes that r's Phase I respondents expected to remove to obtain local version x :

$$\mathsf{ProposalsForVer}(x,r) = \{z \mid \exists q \in \mathsf{Phase1Resp}(r).(\exists p.((z : p : x) \in \mathsf{next}(q)))\}.$$

GMP-2 and GMP-3 require us to prove that each reconfigurer knows exactly which of the processes in $\mathsf{ProposalsForVer}(x,r)$ could have committed invisibly.

Once the reconfiguration algorithm completes, r and the outer processes can begin the exclusion algorithm. If invis is defined, they can begin at the appropriate points in the compressed removal algorithm (line X.1 for r, and line X.2 for the outer processes). Observe that Mgr must henceforth garner responses from a majority of processes before it can commit any removals. We present the final Mgr algorithm when we consider the *join* operation.

5 Correctness of the Reconfiguration Algorithm

Our goal is to show that all successful initiators (Mgr or reconfigurers able to reach the commit phase) determine identical proposals. To do this, will prove that every invisibly committed removal is detectable by every successful reconfigurer. We first show that local views do not diverge so far that majorities need not intersect[14]. To do this, we quantify the possible difference between an initiator's local view and its respondents' by showing

$$\forall q \in \mathsf{Phase1Resp}(r).(\mathsf{ver}(r) - 1 \leq \mathsf{ver}(q) \leq \mathsf{ver}(r) + 1).$$

We will also show that no non-faulty process receives proposals that would force it to skip a version number, thereby guaranteeing a sort of cohesion among the responses an initiator receives to its interrogation :

$$\max_{\pi \in \mathsf{next}(q)} 3rd(\pi) = \mathsf{ver}(q) + 1.$$

Given r, a reconfiguration initiator, $\mathsf{Phase1Resp}(r)$, a majority subset of $\mathsf{Memb}(r)$, $q \in \mathsf{Phase1Resp}(r)$, and $\pi = (z : p : v)$, an element of $\mathsf{next}(q)$, we next show that p cannot succeed in committing any view numbered more than v, and that $v \leq \mathsf{ver}(r) + 2$. In this way, any proposal that has a chance of being committed (*i.e.* one whose initiator receives majority approval) will be known to all subsequent initiators

[14] This is also relevant to GMP-2, for ensuring unique system views requires at most one initiator to be able to obtain responses from a majority of processes in its *local* view.

```
Initiator, r
    [note: Upon full (HiFaulty(r)) Begin Phase I - Interrogation]
    Bcast(r,Memb(r),Interrogate)
    ∀p ∈Memb(r). await (OK(seq(p),next(p)) or faulty_r(p));
    if fewer than μ_r OKs then quit_r.
    [note: Begin Phase II - Proposal];
    Determine(RL_r, invis, v);
    Bcast(r,Memb(r),((RL_r:r : v):(invis,Faulty(r))));
    ∀p ∈Memb(r).(await (OK(p) or faulty_r(p)));
    if fewer than μ_r OKs then quit_r.
    [note: Begin Phase III - Commit]
    remove_r(RL_r);
    Bcast(r,Memb(r),Commit(RL_r):(invis,Faulty(r)));
    seq(r) ← (seq(r),RL_r);
    ver(r) ← ver(r)+1;
    begin Mgr role by removing invis.

Outer Processes, p
    recv(r, p,Interrogate);
    if rank(p)<rank(r) then quit_p.
    send(p, r,OK(seq(p),next(p)));
    ∀q ∈HiFaulty(r).(faulty_p(q));
    next(p) ← (next(p), (? : r :?));
    await (Propose((proc-id : r : v_r):(next-id,F)) or faulty_p(r));
    if faulty_p(r) then exit the protocol.
    if (p ∈ F) then quit_p.
    send(p, r,OK(p));
    ∀q ∈ F.(faulty_p(q));
    next(p) ← (proc-id : r : v_r);
    await (Commit((proc-id : r : v_r):(next-id,F')) or faulty_p(r));
    if faulty_p(r) then exit the protocol.
R.1 if (p ∈ F') then quit_p.
R.2 if v_r ≠ ver(p)
        then remove_p(proc-id);
            ver(p) ← ver(p)+1;
            seq(p) ← (seq(p),proc-id);
    next(p) ← (next-id:r : ver(p) + 1);
    ∀q ∈ F'.(faulty_p(q));
    Mgr ← r.
```

Figure 5: Reconfiguration Algorithm

```
[note: Determine(RL_r, invis, v) : determines a reconfiguration proposal for initiator r]
Begin :
L ← {l | ver(l) = ver(r) + 1};
S ← {s | ver(s) = ver(r) − 1};
case  L ≠ ∅ [note: incomplete installation of version ver(L)]
        begin
        v  ← ver(L);
D.0     RL_r ← (seq(L)−seq(r));
D.1     case  |ProposalsForVer(v+1)| = 0 then  GetNext(invis );
D.2           |ProposalsForVer(v+1)| = 1 then  invis ← ProposalsForVer(v+1);
D.3           else  invis ← GetStable(r, v);
        end
    L = ∅ ∧ S ≠ ∅ [note: incomplete installation of version ver(r)]
        begin
        v  ← ver(r);
        RL_r ← (seq(r)−seq(S));
        case  |ProposalsForVer(v+1)|= 0then  GetNext(invis );
              |ProposalsForVer(v+1)| = 1 then  invis ← ProposalsForVer(v+1);
              else  invis ← GetStable(r, v);
        end
    L = S = ∅
        begin  v  ← ver(r)+1;
D.4     case  |ProposalsForVer(v+1)| = 0 then  RL_r ← Mgr;
D.5           |ProposalsForVer(v+1)| = 1 then  RL_r ← ProposalsForVer(v+1);
D.6           else  RL_r ← GetStable(r, v);
        GetNext(invis );
        end
End.
```

```
[note: GetStable(r, ver) : determines the one process in ProposalsForVer(ver,r)]
[note: whose removal could have been committed invisibly to r]
Begin :
Proposers_r  ← {p | ∃q ∈ Phase1Resp(r), z_p ∈ ProposalsForVer(ver).((z_p : p : ver) ∈ next(q))};
let  p ∈ Pr_r such that (∀p' ∈Proposers_r.(rank(p)<rank(p')));
[note: i.e. p is the lowest-ranked process to have proposed version ver]
let  π_p such that π_p = (z_p : p : ver);
GetStable  ← z_p;
End.
```

Figure 6: Procedures *Determine*(RL_r, invis, v) and *GetStable*(r, ver) of Reconfiguration

reaching the commit phase; we will have ensured that, until a reconfiguration completes, majority subsets of local versions must intersect.

To show correctness with respect to GMP-2 and GMP-3 entirely, we will also need to show that in each of the sets of r-detectable proposals for version x, $\mathsf{ProposalsForVer}(x,r)$, there is at most one process whose removal may have been committed invisibly to r, and that r can determine which process it is. In particular we first show

- $\forall r.|\mathsf{ProposalsForVer}(x,r)| \leq 2$, and
- $(|\mathsf{ProposalsForVer}(x,r)| = 2 \wedge |\mathsf{ProposalsForVer}(x,r')| = 2) \Rightarrow$

$$\mathsf{ProposalsForVer}(x,r) = \mathsf{ProposalsForVer}(x,r').$$

The major work is in showing that only one of the two r-detectable proposals *could* have been committed, and that r can determine which of the two it is. From these facts a weakened version of the safety conditions follows :

$$\forall p, q \in \mathsf{Phase1Resp}(r).(\mathsf{ver}(p) = \mathsf{ver}(q) \Leftrightarrow \mathsf{seq}(p) = \mathsf{seq}(q)) \tag{1}$$

In what follows, let L be the subset of $\mathsf{Phase1Resp}(r)$ reporting the *longest* sequence or remove events, and S be the subset with the *shortest* sequence. Let "$?x$" denote the invitation (if removal) or interrogation/proposal (if reconfiguration) messages for the xth intended system view, Sys^x , and "$!x$" denote the commit message (whether removal or reconfiguration) for Sys^x .

Proposition 5.1 *If r is a reconfiguration initiator then*

$$\forall q \in \mathsf{Phase1Resp}(r).(\mathsf{ver}(r) - 1 \leq \mathsf{ver}(q) \leq \mathsf{ver}(r) + 1).$$

Proof Let $\mathsf{ver}(r) = x$ and p be the process responsible for r installing Memb_r^x. While Sys^x may not be fully defined, r has installed it locally. Suppose that some $s \in \mathsf{Phase1Resp}(r)$ has $\mathsf{ver}(s) < x - 1$. Then s has neither received nor responded to p's "$?x$", so p believes $faulty_p(s)$. Upon receipt of p's commit message, "$!x$", r also believes $faulty_r(s)$ and will receive no further messages from s.

On the other hand, suppose some $l \in \mathsf{Phase1Resp}(r)$ responds with $\mathsf{ver}(l) > x + 1$, and let p' be responsible for installing $\mathsf{ver}(l)$. Because $\mathsf{ver}(r) = x$, r has neither received nor responded to p''s "$?\mathsf{ver}(l)$", resulting in $faulty_{p'}(r)$, and, upon l's receipt of "$!\mathsf{ver}(l)$", $faulty_l(r)$. In such a situation, l would not receive or respond to r's interrogation. ∎

Definition Given process p, Sys^x is *p-defined* (along consistent cut c) if

$$\bigwedge_{q \in \mathsf{Memb}(p)} (K_p \mathsf{ver}(q) \geq x) \vee (faulty_p(q)). \tag{2}$$

That is, from the point of view of process p, Sys^x is (or has been) defined. Of course, Sys^x may not be technically defined as some process q, which p believes faulty, may have $\mathsf{ver}(q) < x$ and still be functioning. With respect to a reconfiguration initiator, r, Sys^x is r-defined when every process in $\mathsf{Phase1Resp}(r)$ reports, in Phase I, a version number at least as large as x. At the end of Phase I, r believes all those in $\overline{\mathsf{Phase1Resp}(r)}$ faulty.

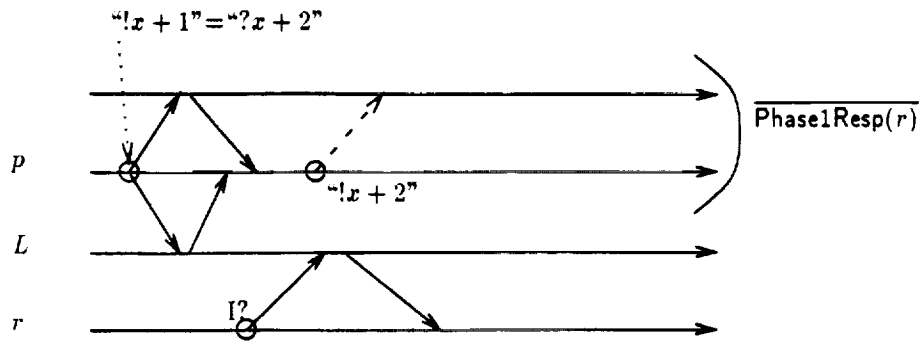

Figure 7: Bounding Invisible Commits

Proposition 5.2 *Let r be a reconfiguration initiator. Then r proposes version x only if* Sys^{x-1} *is the last r-defined system view (once r has finished Phase I).*

Proof With reference to procedure *Determine*,

- When $L \neq \emptyset$, r proposes version number $\mathsf{ver}(L) = \mathsf{ver}(r) + 1$, and while it may also be the case that $S \neq \emptyset$, it is not difficult to see that $faulty_L(S)$ holds resulting in $faulty_r(S)$ at the end of Phase I and $\mathsf{Sys}^{\mathsf{ver}(r)}$ being r-defined there.
- When $L = \emptyset \wedge S \neq \emptyset$, $\mathsf{Sys}^{\mathsf{ver}(S)}$ is the last r-defined system view and r proposes version number $\mathsf{ver}(r) = \mathsf{ver}(S) + 1$.
- When $L = S = \emptyset$, $\mathsf{Sys}^{\mathsf{ver}(r)}$ is the last r-defined system view and r proposes version $\mathsf{ver}(r) + 1$.

■

Proposition 5.3 *No non-faulty process receives a proposal that would force it to skip a local view :*

$$\forall q \in \mathsf{Phase1Resp}(r).(\max_{\pi \in \mathsf{next}(q)} 3rd(\pi) = \mathsf{ver}(q) + 1).$$

Proof Let $\mathsf{ver}(q) = x$ and suppose r proposes $\pi = (z : r : x + 2)$ and $\pi \in \mathsf{next}(q)$. Then it is not possible for r to be the same co-ordinator as the one responsible for installing Memb_q^x because the FIFO channel assumption forces q to receive "$?x + 1$" before "$!x + 1$" and "$!x + 1$" before "$?x + 2$". But then $\mathsf{ver}(q) = x + 1$.

So suppose r is a reconfiguration initiator. Proposition 5.2 shows that r proposes version number $x + 2$ if and only if r detects Sys^{x+1} as the last r-defined system view. This, we have already noted, means $\forall p \in \mathsf{Phase1Resp}(r).(\mathsf{ver}(p) \geq x+1)$. We surmise, then, that r did not receive q's response to its interrogation. In this case, $q \in \mathsf{Faulty}(r)$, and upon receipt of r's proposal executes $quit_q$ (R.1) before it updates $\mathsf{next}(q)$.

■

The cohesion of Phase I responses is important in the next proposition.

If r is successful in obtaining a majority of responses from the processes in Memb_r^x, Proposition 5.1 tells us that the largest version number observed among r's respondents is $x + 1$; thus, $\forall l \in L.(\mathsf{ver}(l) \leq x + 1)$. So suppose $\mathsf{ver}(L) = x + 1$. Then every $l \in L$ has responded to "$?x + 2$". Moreover, (all) processes in $\overline{\mathsf{Phase1Resp}(r)}$ may also have done so. It is possible that L and $\overline{\mathsf{Phase1Resp}(r)}$ together may suffice to

form the requisite majority, μ^{x+1}, to commit version $x+2$ (See Figure 7). Fortunately, we can bound the divergence in the system by showing that if r has obtained responses from a majority of processes in Memb_r^x, no other process can (concurrently) succeed in committing local versions numbered higher than $x+2$.

Proposition 5.4 *Let r be a reconfiguration initiator and let $L \subseteq \mathsf{Phase1Resp}(r)$ have the largest local version number. Given $l \in L$, let $\pi = (z : p : v) \in \mathsf{next}(l)$. Then while p may succeed in committing the removal of z invisibly to r, p cannot succeed in installing any view numbered greater than $\mathsf{ver}(l)+1$.*

Proof From Proposition 5.1 we know $\mathsf{ver}(r) \leq \mathsf{ver}(l) \leq \mathsf{ver}(r)+1$. Proposition 5.3 gives $v \leq \mathsf{ver}(l)+1 = \mathsf{ver}(r)+2$. For the worst case, take $v = \mathsf{ver}(r)+2$. Then process p can succeed in committing view $v+1$ if and only if $|\overline{\mathsf{Phase1Resp}(r)} - \{z\}| \geq \mu^v$. Noting that $\mathsf{Sys}^x = \mathsf{Memb}_r^x = \left\{\mathsf{Phase1Resp}(r), \overline{\mathsf{Phase1Resp}(r)}\right\}$, that $z \in \overline{\mathsf{Phase1Resp}(r)}$, and that $\mu^x - 1 = \mu^{x+2}$ for each x, then p succeeds if and only if

$$(|\ \overline{\mathsf{Phase1Resp}(r)} - \{z\}\ | \geq \mu^v) \Leftrightarrow (|\ \overline{\mathsf{Phase1Resp}(r)}\ | \geq \mu^v + 1)$$

$$\Leftrightarrow (|\ \overline{\mathsf{Phase1Resp}(r)}\ | \geq (\mu^{\mathsf{ver}(r)} - 1) + 1) = \mu^{\mathsf{ver}(r)}.$$

But $|\overline{\mathsf{Phase1Resp}(r)}| \geq \mu^{\mathsf{ver}(r)}$ is impossible. ∎

Recall the definition of the sets $\mathsf{ProposalsForVer}(x,r)$:

$$\mathsf{ProposalsForVer}(x,r) = \{z \mid \exists q \in \mathsf{Phase1Resp}(r).(\exists p.((z : p : x) \in \mathsf{next}(q)))\}.$$

It remains to consider how r can determine which (one) of the elements in these sets could have been invisibly committed. This is important in determining invis (when either $L \neq \emptyset$ or $L = \emptyset \wedge S \neq \emptyset$) and in determining RL_r (when $L = S = \emptyset$).

To elucidate, suppose Sys^{x-1} is the last r-defined view. Intuitively, if Sys^{x-1} was committed with an attendant proposal for Sys^x (*i.e.* the condensed algorithm applied), then $\mathsf{ProposalsForVer}(x,r)$ is that proposal and $|\mathsf{ProposalsForVer}(x,r)| = 1$. However, it may be the case that, while there were no plans for future removals when **Mgr** broadcast the commit message for Sys^{x-1} , at some later time, **Mgr** began an exclusion algorithm to form the xth system view. If, during that same interval, a process had begun reconfiguration, it is possible that it may not receive any Phase I responses indicating **Mgr**'s plans for Sys^x . In such a case, this reconfigurer would propose **Mgr**'s removal for version x. A subsequent reconfigurer may then get responses indicating both of these proposals.

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

Proposition 5.5 *Let r be a reconfiguration initiator proposing version x. Then,*

$$\forall x > 0.(|\ \mathsf{ProposalsForVer}(x,r)\ | \leq 2).$$

Proof Suppose **Mgr** succeeded in inviting a majority of processes to install version x (that is, a majority of Sys^{x-1} received "?x" from **Mgr**). Let S denote the set of processes receiving "?x" from **Mgr**. Before any reconfiguration attempts take place, there is only one element in the general class, $\mathsf{ProposalsForVer}(x)$. Now, the 'first' reconfigurer, r_1, obtaining a majority of responses in Phase I[15] must have $|\mathsf{ProposalsForVer}(x,r_1)| = 1$

[15] From the majority property, it is not difficult to see that 'first', 'second' etc. are well-defined here.

because Phase1Resp(r_1) must intersect S. From line D.5 , r_1 propagates this. Similar reasoning tells us that the proposal choices for the second, third, and so on, reconfigurers that complete Phase I are identical. In this way, the only proposals made for version x propagate Mgr 's proposal, and |ProposalsForVer(x,r)|= 1 for every r proposing version x.

So suppose the set S is not a majority of Sys^{x-1} . Let r' be a reconfigurer for which Phase1Resp(r') is a majority, and suppose no process in Phase1Resp(r') has heard of Mgr 's proposal. Then r' proposes to remove Mgr (line D.4). In general, all reconfigurers not detecting Mgr 's proposal (directly or by propagation) propose to remove Mgr , and all reconfigurers detecting *only* Mgr 's proposal propagate it. In this way, the general class, ProposalsForVer(x), can contain two elements. The first reconfigurer to detect both these proposals calls procedure *Determine* which chooses exactly one of them to propagate, thereby introducing no further proposals. ∎

Corollary 5.1 *Let r and r' be reconfigurers proposing version x. Then*

$$((|\ \mathsf{ProposalsForVer}(x,r)\ |= 2) \wedge (|\ \mathsf{ProposalsForVer}(x,r')\ |= 2)) \Rightarrow$$

$$\mathsf{ProposalsForVer}(x,r) = \mathsf{ProposalsForVer}(x,r').$$

Proof All reconfigurers either propagate Mgr 's proposal for version x, which is unique, or propose Mgr '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 plays a crucial role in simplifying the full correctness proofs in the next section.

Proposition 5.6 *Let r be a reconfiguration initiator. If* |ProposalsForVer(x,r)|= 2, *r can distinguish which of the two proposals could not have been committed invisibly.*

Proof Let r be the first process for which Phase1Resp(r) is a majority of Memb(r) and such that |ProposalsForVer(x,r)|=2. Let p and p' be such that ($z : p : x$) and (Mgr $: p' : x$) are found in the responses to r's interrogation. Without loss of generality, let rank(p)>rank(p') and consider the following cases :

1. p = Mgr . From the proof of Proposition 5.5, we know Mgr 's proposal to remove z did not reach a majority of Sys^{x-1} , and Mgr could not have succeeded in committing z's removal.

2. $p \neq$ Mgr . Since p and p' were both able to propose views, Phase1Resp(p) and Phase1Resp(p') must intersect. If Phase2Resp(p) and Phase1Resp(p') intersect then ($z : p : x$) is known to p' resulting in $z \in$ ProposalsForVer(x,p'). By hypothesis, r is the first process to see both proposals, so ProposalsForVer(x,p')={z}. In this case p' is forced to propagate ($z : p : x$) (line D.5) and cannot have proposed (Mgr $: p' : x$).

   So it must be that Phase2Resp(p) and Phase1Resp(p') *do not* intersect and r deduces that Phase2Resp(p) could not have been a majority. It is therefore impossible for p to have committed z's removal invisibly to r (and p').

An analogous argument applies when rank(p')>rank(p)

∎

Proposition 5.6 shows that *GetStable* correctly chooses the only proposal for a given version number that *could* have been invisibly committed.

On a side note, if $|\mathsf{ProposalsForVer}(x,r)| = 1$ then propagating $\mathsf{ProposalsForVer}(x,r)$ is safe as no other higher-ranked process can obtain the majority required to partially commit a different version x. For the same reasons, if $|\mathsf{ProposalsForVer}(x,r)| = 0$, proposing Mgr's removal is also safe.

Definition A proposal is *stably-defined* if its initiator could possibly have reached the commit stage; that is, given $\pi = (z : p : x)$, if $p = \mathsf{Mgr}$, then p obtained responses from a majority of processes in Sys^{x-1}, and if $p \neq \mathsf{Mgr}$, then $\mathsf{Phase2Resp}(p)$ is a majority subset of $\mathsf{Memb}(p)$ and $x - 1 \leq \mathsf{ver}(p) \leq x$.

Stably-defined proposals are exactly the proposals that any reconfigurer must view as possibly committed invisibly.

Corollary 5.2 *All stably-defined proposals for the same version number are identical.*

Proof Proposition 5.6 proves that any reconfigurer reaching its proposal stage knows exactly which of the two proposals for a given version number is not stably-defined. Procedures *Determine* and *GetStable* 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 that version. ∎

Theorem 5.1 (Identical Local Views - Weak) *Let r be a reconfiguration initiator. Then*

$$\forall p, q \in \mathsf{Phase1Resp}(r).(\mathsf{ver}(p) = \mathsf{ver}(q) \Rightarrow \mathsf{seq}(p) = \mathsf{seq}(q)).$$

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

Remarks

Our algorithms ensure that the state to which the system finally reconfigures represents the *cumulative system progress*. It accounts for any previous updates (and reconfigurations) that could (and may) have been only partially successful, and makes them stable. With respect to an interrupted commit, say of Sys^x, the xth system view $(\mathsf{Sys}^{x-1} - \{z\})$ does not exist until r succeeds in broadcasting its reconfiguration commit messages.

To see that the new Mgr is unique, consider a process, p, that has received an interrogation from r. It disconnects its incoming channel with every process in $\mathsf{HiFaulty}(r)$, and therefore ceases to receive messages, particularly messages relating to exclusion or reconfiguration, from processes in $\mathsf{HiFaulty}(r)$. Thus, p immediately begins to believe that r is the highest ranking non-faulty process.

Finally, within certain limits, the reconfiguration proposal RL_r may be more than just a single process. Its size is a function of the current size of the system view and must guarantee that majority subsets of $\mathsf{Memb}(r)$ and $\mathsf{Memb}(r) - \{\mathsf{RL}_r\}$ intersect.

6 Correctness Proofs

Proposition 6.1 *The Full Algorithm satisfies GMP-0.*

Proof Follows immediately from the initial assumptions. ∎

Proposition 6.2 *The Full Algorithm satisfies GMP-1.*

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

1. Commit(q):Contingent(next-id:L) from Mgr, in which case the Remove Algorithm gives either

   (a) $recv(\mathsf{Mgr}, p, \text{Exclude}(q)) \rightarrow faulty_p(q) \rightarrow recv(\mathsf{Mgr}, p, (q):\text{L}) \rightarrow remove_p(q)$,
   or, if the condensed algorithm can be applied,

   (b) $recv(\mathsf{Mgr}, p, (q'):(q':\text{L})) \rightarrow ((\forall l \in \text{L}).faulty_p(l)) \rightarrow recv(\mathsf{Mgr}, p, (q:\text{L}')) \rightarrow remove_p(q)$.

2. Commit(RL_r:r : x):Contingent(invis:Faulty(r)) from some reconfiguration initiator, r. In this case, observe that proposals always precede commit messages and that p executes $faulty_p(\mathsf{RL}_r)$ upon receipt of r's proposal, Propose(RL_r:r : x).

∎

To prove that the Full Algorithm satisfies GMP-2 and GMP-3, we rely heavily on Theorem 5.1. To prove GMP-2, we will exhibit the cuts c_x and show uniqueness of the system view; GMP-3 is a simple corollary of the theorem.

Theorem 6.1 *The Full Algorithm satisfies GMP-2.*

Proof Let r_x be the process responsible for *completing* the installation of Sys^{x-1} and q be the process removed from Sys^{x-1} in obtaining Sys^x . Define the cut c_x as :

$$c_x[p] = \begin{cases} recv(r_x, p, \text{Commit}(q)) & remove_p(q) - recv(r_x, p, \text{Commit}(q)) \\ remove_p(q) & recv(r_x, p, \text{Commit}(q)) - remove_p(q) \\ quit_p & \text{otherwise} \end{cases} \tag{3}$$

It is easy to see that c_x is consistent and that $c_x \ll c_{x+1}$. We now show $\mathsf{Sys}(c_x, \mathsf{Memb}(p, c_x))$=$\mathsf{Memb}(p, c_x)$.

From GMP-0 and Proposition 6.1, Proc=Memb_p^0 so $\mathsf{Memb}(p, c_0) = \mathsf{Proc}$, and $\mathsf{Sys}(c_0, \mathsf{Memb}(p, c_0)) = \mathsf{Memb}(p, c_0)$.

From Corollary 5.2, we know that all stably-defined proposals for the same version number are identical. Then

$$\forall p \in \mathsf{Sys}^{x-1} \cap \mathcal{UP}(c_x).(\mathsf{Memb}_p^x = \mathsf{Memb}_p^{x-1} - \{q\})$$

By definition, $\mathsf{Memb}_p^x = \mathsf{Memb}(p, c_x)$ for all p, and this leaves $\mathsf{Sys}(c_x, \mathsf{Memb}(p, c_x))$=$\mathsf{Memb}(p, c_x)$.

Uniqueness of the system views follows from Corollary 5.2 and the majority requirement for any process hoping to install a new system view. ∎

Theorem 6.2 *The Full Algorithm satisfies GMP-3.*

Proof Recall that successful initiators are those able to reach the commit phase, and that stably-defined proposals are those issued by successful initiators. Corollary 5.2 shows that all stably-defined proposals for the same version number are identical. Thus, Memb_p^x=Memb_q^x for each p and q. ∎

Proposition 6.3 *The Full Algorithm satisfies GMP-4.*

Proof Processes only update their local views with the set difference operation which never adds processes. ■

Proposition 6.4 *The Full Algorithm satisfies GMP-5.*

Proof No requests made by processes for a particular **Mgr** to initiate the exclusion algorithm are 'lost' when a new **Mgr** is installed. An outer process's, p's, local beliefs of $faulty_p(q)$ are propagated by system property F2 during reconfiguration. ■

7 The Join Procedure

The *Join* procedure is a simple variation of the *Remove* procedure, with restrictions regarding majority approval. **Mgr** initiates the join algorithm for process p when it becomes aware of p's desire to join the group. Recall that 'recovered' processes are treated as new, and different process instances.

In the last section we saw that correctness with respect to both GMP-2 and GMP-3 hinges on majority subsets of 'neighboring' views (Memb_p^x and Memb_r^{x+1}, for $x \geq 0$ and any processes r and p) intersecting. When this was the case, we ensured both uniqueness of system views and complete detection of invisible commits. Toward proving a general result, let S be an arbitrary set and define the cardinality of a majority subset of S as $\mu(S) \equiv (\lfloor \frac{|S|}{2} \rfloor + 1)$. Then given sets S and S', the following facts underly the correctness of our algorithms :

Fact 7.1 *For all sets S, if $|S|$ is even, then $2\mu(S) = |S| + 2$.*

Fact 7.2 *For all sets S, if $|S|$ is odd, then $2\mu(S) = |S| + 1$.*

Fact 7.3 *For all sets S and S', if $|S'|=|S|+1$, and $\mu(S') = \mu(S) + 1$ then $|S'|$ is even.*

Proposition 7.1 *For all sets S and S', if $|S'|=|S|+1$, then $\mu(S) + \mu(S') > |S'|$.*

Proof If $\mu(S') = \mu(S)$ then $\mu(S) + \mu(S') = 2\mu(S')$ and $2\mu(S') > |S'|$ by definition. Otherwise if $\mu(S') = \mu(S) + 1$ then Fact 7.3 tells us that $|S'|$ is even. From Fact 7.1 we know $2\mu(S') = |S'| + 2$. Therefore $\mu(S) + \mu(S') = 2\mu(S') - 1 = |S'| + 1$, giving $\mu(S) + \mu(S') > |S'|$. ■

With respect to our algorithms, this means that majority subsets of neighboring views will intersect : each invocation of our algorithm can change the existing system view by either removing or adding *exactly one* process. In this way, either

$$\text{Add} : \mathsf{Memb}_p^x \subset \mathsf{Memb}_r^{x+1} \text{ and } |\mathsf{Memb}_r^{x+1}| = |\mathsf{Memb}_p^x| + 1, \text{or}$$

$$\text{Remove} : \mathsf{Memb}_r^{x+1} \subset \mathsf{Memb}_p^x \text{ and } |\mathsf{Memb}_p^x| = |\mathsf{Memb}_r^{x+1}| + 1.$$

7.1 The Final Algorithm

For the final algorithm, we alter both the invitation and commit messages to include the desired operation, 'add' or 'remove'. For example, Invite(add(q)), and Commit(remove(p)). Similarly, **next**(p) and **seq**(p) will prepend the relevant operation to each process identifier. The reconfiguration proposal message will also indicate the desired operation. **ver**(p) will continue to reflect the instance (or ordinality) of **Memb**(p). Finally, the local sets **Recovered**(p) are analogous to the sets **Faulty**(p).

Procedures *Determine* and *GetStable* are as in Section 4.5.

```
Update Algorithm - Mgr
Begin :
while true do
    begin
    await (Recovered(Mgr) ≠ ∅ or Faulty(Mgr) ≠ ∅);
    if Recovered(Mgr)≠ ∅
        then proc-id ← delete (Recovered(Mgr));
            op ← 'add';
        else proc-id ← delete (Faulty(Mgr));
            op ← 'remove';
    while (proc-id≠ nil-id) do
        begin
        Bcast(Mgr,Memb(Mgr),Invite(op(proc-id)));
        ∀p ∈ Memb(Mgr).(await (OK(p) or faulty_Mgr(p)));
        if fewer than μ_Mgr OKs then quit_Mgr.
        if op='add'
            then add_Mgr(proc-id);
            else remove_Mgr(proc-id);
        GetNext(next-id,next-op);
        ver(Mgr) ← ver(Mgr)+1;
        Contingencies ← (next-op(next-id):Faulty(Mgr):Recovered(Mgr)));
        Bcast(mgr,Memb(Mgr),Commit(op(proc-id)):Contingencies);
        proc-id ← next-id;
        op ← next-op;
        end ;
    end ;
End.
```

Figure 8: The Final Update Algorithm - Mgr

```
Update Algorithm - Outer Processes
Begin :
recv(Mgr , p,Invite(op(proc-id)));
if op='add'
    then operating_p(proc-id)
    else faulty_p(proc-id);
repeat send(p, Mgr ,OK(p))
    await (Commit(op(proc-id)):Cgt(next-op(next-id):F:R) or faulty_p(Mgr ));
    if faulty_p(Mgr ) then exit.
    if (p ∈ F) then quit_p.
    if next-op = 'add'
        then operating_p(next-id)
        else faulty_p(next-id);
    ∀f ∈ F.(faulty_p(f));
    ∀r ∈ R.(operating_p(r));
    if op='add'
        then add_p(proc-id);
        else remove_p(proc-id);
    ver(p) ← ver(p)+1;
    proc-id — next-id;
    op — next-op;
until (proc-id=nil-id);
End.
```

Figure 9: The Final Update Algorithm - Outer Processes

```
Reconfiguration Algorithm - Initiator, r
    [note: Phase I : Upon full (HiFaulty(r))]
    Bcast(r,Memb(r),Interrogate)
    ∀p ∈Memb(r). await (OK(seq(p),next(p)) or faulty_r(p));
    if fewer than μ_r OKs then quit_r.
    [note: Phase II]
    Determine(RL_r, invis, v);
    Bcast(r,Memb(r),(RL_r:r : v):(invis,Faulty(r))));
    ∀p ∈Memb(r).(await (OK(p) or faulty_r(p)));
    if fewer than μ_r OKs then quit_r.
    [note: Phase III]
    if op='add'
        then add_r(RL_r)
        else remove_r(RL_r);
    Bcast(r,Memb(r),Commit(RL_r):(invis,Faulty(r)));
    seq(r) ← (seq(r),RL_r);
    ver(r) ← ver(r)+1;
    begin Mgr role with relevant operation on invis.

Reconfiguration Algorithm - Outer Processes, p
    recv(r, p,Interrogate);
    if rank(r)<rank(p) then quit_p.
    send(p, r,OK(seq(p),next(p)));
    ∀q ∈HiFaulty(r).(faulty_p(q));
    next(p) ← (next(p), (? : r :?));
    await (Propose((op(proc-id):r : v_r):(next-op(next-id),F)) or faulty_p(r));
    if faulty_p(r) then exit the protocol.
    if faulty_r(p) then quit_p.
    send(p, r,OK(p));
    ∀q ∈ F.(faulty_p(q));
    next(p) ← (op(proc-id):r : v_r);
    await (Commit((op(proc-id):r : v_r):(next-op(next-id),F')) or faulty_p(r));
    if faulty_p(r) then exit the protocol.
    if faulty_r(p) then quit_p.
    if v_r ≠ ver(p)
        then if op='add'
            then add_p(proc-id)
            else remove_p(proc-id);
        ver(p) ← ver(p)+1;
        seq(p) ← (seq(p),op(proc-id));
    next(p) ← (next-op(next-id):r : ver(p) + 1);
    ∀q ∈ F'.(faulty_p(q));
    Mgr ← r.
```


Figure 10: Final Reconfiguration Algorithm

Remarks

While Mgr needs responses from a majority of processes to safeguard both GMP-2 and GMP-3 (line FA.1) there is a particular situation in which it is permissible to continue without a technical majority. Consider the reasons that Mgr may observe $faulty_{\mathsf{Mgr}}(q)$ while awaiting responses. It may be that q concurrently believes Mgr faulty and is not responding, or it may be that q (or the connection between the two processes) failed. Should Mgr 'time-out' on q, and before finishing its await stage, receive notification of q's subsequent 'recovery', Mgr can, given certain provisos, safely interpret this as q's affirmative response. The provisos concern the actual manner and semantics in which a process's recovery becomes known.

7.2 Complexity Analysis

The sequence and timing of failures affect our algorithm's performance in terms of message complexity. We consider the 'worst' and 'best' case complexity for our protocol to install a new system view. We also quantify the gain achievable when we can use the compressed update algorithm.

Define $n_x \equiv |\mathsf{Sys}^x|$, and τ_x to be the number of tolerable failures in Sys^x ; $\tau_x \equiv (\lceil \frac{n_x}{2} \rceil + 1)$. Then the "worst case" to install the $(x+1)^{st}$ system view occurs when there are τ_x successive failed (or aborted) reconfigurations. This results in

$$\sum_{y=1}^{\tau_x}((n_x - 1) - (y-1))) + 4((n_x - 1) - (y-1)) = 5n_x\tau_x - \frac{5}{2}(\tau_x)^2 - \frac{9}{2}\tau_x = O((|\mathsf{Sys}^x|)^2)$$

messages. Fortunately, this specific composition and timing of failures occurs with very low probability.

There are three 'best case' scenarios in which a successive view can be installed : by Mgr using the straight-forward two-phase update algorithm, by Mgr using the compressed update algorithm, and by one successful reconfigurer. In the first case, at most $3n_x - 5$ messages are required; in the second, at most $2n_x - 3$; in the third, at most $5n_x - 9$.

Finally, if we can take advantage of the condensed algorithm (if failures are not spaced 'too far' apart), we save substantially in message complexity. For $n-1$ successive failure updates, none of which are Mgr, we require

$$(n-1) + 2\sum_{x=2}^{n-1}(n-x) = (n-1) + 2n(n-2) - 2\sum_{x=2}^{n-1} = n^2 - 2n - 1 = (n-1)^2 \approx n^2$$

messages, averaging to $n-1$ messages per exclusion. A standard two-phase algorithm would require an additional $\frac{n}{2} - 1$ messages per exclusion, on the average.

In all of these cases, actual failures may reduce the number of response messages and thereafter the number in the broadcast.

7.3 Optimality Results

Our GMP protocol combines two-phase (basic update) and three-phase (reconfiguration) commit protocols. Neither one-phase (*i.e.* a simple broadcast by a unique coordinator) nor two-phase protocols are sufficient for solving GMP. This is similar to the result in [20] in which it is shown that a three-phase commit algorithm is necessary in maintaining the consistency of a distributed database. We now give an intuitive proof of this for GMP.

It is not difficult to show that a one-phase algorithm cannot guarantee GMP-3.

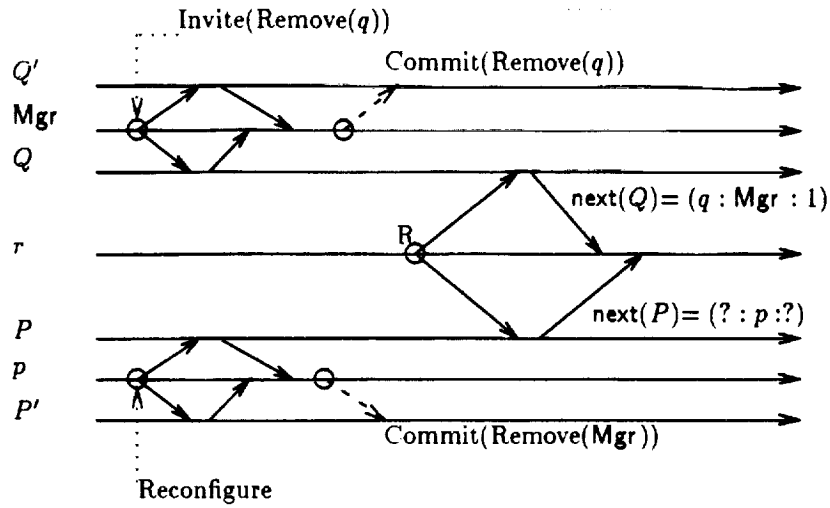

Figure 11: Inability to Determine Invisible Commits

Claim 7.1 *A one-phase update algorithm cannot solve GMP when the coordinator can fail.*

Proof Let R and S partition Proc, and let $r \in R$, and Mgr $\in S$. Suppose the following are process histories : $start_R \rightarrow faulty_R(\mathsf{Mgr})$, and $start_S \rightarrow faulty_S(r)$. Now, r's reconfiguration commit message (removing Mgr) can only be received by processes in R and Mgr 's exclusion commit message (removing r) can only be received by processes in S. Then

$$\mathsf{Memb}_R^1 = \mathsf{Proc} - \{\mathsf{Mgr}\} \neq \mathsf{Proc} - \{r\} = \mathsf{Memb}_S^1,$$

violating GMP-3. ∎

To show that a two-phase algorithm is incapable of satisfying GMP, we exhibit a situation in which it is impossible for a reconfigurer, knowing that only one of two proposals could possibly have been committed invisibly, to determine which one it is. If it chooses the wrong one to propagate, GMP-3 is violated.

Claim 7.2 *A two-phase reconfiguration algorithm cannot solve GMP when the coordinator can fail.*

Proof Consider Figure 11 in which both r and p are reconfigurers and neither Q nor P are majority subsets of Sys^0 . Let next(p) be a triple indicating the process p plans to remove next, upon which other process's command, and which local view number results. Upon completion of its Phase I, r knows that exactly one of Mgr and p *could* have been successful in obtaining the requisite majority of responses, but it has no way of determining which, if any, of the two did.

Let Phase1Resp(r) be the set of processes responding to r's Phase I reconfiguration message (In Figure 11, Phase1Resp(r)= $Q \cup P \cup \{r\}$). Then r can envision one case in which all of $\overline{\mathsf{Phase1Resp}(r)}$ (*i.e.*$Q' \cup P'$) responded to Mgr, allowing it to succeed, and another situation in which they responded to p, fulfilling its majority requirement. Thus, r does not know whether to propagate Mgr 's proposal or p's. If it guesses incorrectly, it violates GMP-3. ∎

8 Conclusion

We have presented a formal specification of process group membership as it relates to failure detection in asynchronous systems. The need for formalism in this area (and others) is amply demonstrated by reviewing the recent literature, as there are many different problems being solved, each of which claim to be 'The Group Membership Problem'. Not surprisingly, some of these loose specifications admit trivial, and even incorrect, solutions. We developed a solution to our Group Membership Problem, analyzing it in terms of both process knowledge and message complexity. We used the former to show that the Fischer-Lynch-Paterson impossibility result does not apply to this work. The latter is used to compare our solution to solutions of similar problems. In this regard, our solution is an order of magnitude cheaper than ([15], [5]). Our solution also improves upon others' ([6], [4]) by handling a continuous stream of failures and recoveries (provided a majority of processes are not seen to fail during any one instance of the algorithm).

We have formally shown the solution satisfies our problem specification. Moreover, while we have shown that a three-phase protocol is necessary for reconfiguration, we are currently investigating an optimization to our algorithm that would allow a process, in specific circumstances, to take advantage of previous communication phases initiated by other processes. Thus, similar to the way we compressed the update algorithm, we would pare down required communication when failures of reconfiguration initiators are continuous.

We emphasize that our particular formulation reflects our application's requirements for group membership; how an asynchronous failure detection mechanism uses process groups and the meaning attached to membership in a process group. Other applications will have different restrictions, and one could weaken or strengthen the definition of GMP in a number of ways. For example, by not requiring processes to be members of their own local views, we can create a hierarchical management service. The group might be a set of clients with exclusion from it would modelling the end of that client's need for the service. Similarly, we need not require the sets S_x (used in defining Sys^x) to be unique; some applications (for example the Deceit File System [19] and El Abbadi and Toueg's database consistency algorithm [1]) may wish to allow partitions to exist and have them dealt with at a different level. Additionally, requiring every locally committed view to exist as a system view (our GMP-3) is a restriction inherited from the fact that processes may take external actions that reflect a particular group composition.

Acknowledgements

We would like to thank Keith Marzullo, Vassos Hadzilacos, Flaviu Cristian, Tushar Chandra, and Sam Toueg for their many helpful comments and observations.

References

[1] A. El Abbadi and S. Toueg. Availability in Partitioned Replicated Databases. In *Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 240–251, Cambridge, MA, March 1986.

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

[3] K. P. Birman and T. A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems". In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 1987.

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

[5] S. A. Bruso. A Failure Detection and Notification Protocol for Distributed Computing Systems. In *Proceedings of the IEEE 5th International Conference On Distributed Computing Systems*, pages 116–123, May 1985.

[6] J. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

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

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

[9] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the Association for Computing Machinery*, 35(2):288–323, April 1988.

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

[11] J.Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. In *Proceedings of the Third A.C.M. Symposium on Principles of Distributed Computing*, pages 50–61, 1984.

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

[13] K. Marzullo. Personal Communication. January, 1991.

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

[15] S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol Based on Partial Order. Technical report, University of Arizona, 1990.

[16] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Processor Membership in Asynchronous Distributed Systems. University of California at Santa Barbara, Extended Abstract, 1990.

[17] P. Panangaden and K. Taylor. Concurrent Common Knowledge: A New Definition of Agreement for Asynchronous Systems. In *Proceedings of the Seventh Annual A.C.M. Symposium on Principles of Distributed Computing*. A.C.M., 1988.

[18] A. Ricciardi. Completeness of a Tense Logic for Asynchronous Systems. In *Proceedings of the 1990 International Conference of Computer Science Logic*, 1990. Heidelberg, Germany.

[19] A. Siegel, K. Birman, and K. Marzullo. Deceit: A Flexible Distributed File System. In *Summer 1990 USENIX Conference*, pages 51–61, Anaheim, CA, June 1990. USENIX Association.

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

[21] K. E. Taylor. *Knowledge and Inhibition in Asynchronous Distributed Systems*. PhD thesis, Cornell University, July 1990.

9 Appendix - Epistemic Analysis of GMP

GMP's specification can be phrased in terms of process knowledge. GMP-3 requires every process's local version x to be the same as every other process's local version x. GMP-2 says that there must be some point in every execution when the xth system view exists, resulting in a causally constrained 'consensus'. Thus, when p commits version x it knows, eventually Memb_p^x will be the xth system view. This can be phrased as

$$(\mathsf{ver}(p) = x) \Rightarrow K_p \Diamond (\mathsf{Sys}^x = \mathsf{Memb}_p^x)$$

Define the formula $IsSysView(x)$ to hold exactly when Sys^x is defined :

$$IsSysView(x) \equiv \bigwedge_p ((\mathsf{ver}(p) = x) \wedge (\bigwedge_q ((\mathsf{Memb}_p^x = \mathsf{Memb}_q^x) \vee down(q))) \vee down(p))$$

Noting that $IsSysView(x) \Rightarrow \bigwedge_p (\mathsf{ver}(p) = x)$, and that $(\bigwedge_i (\phi_i \Rightarrow \psi_i)) \Rightarrow (\bigwedge_i \phi_i \Rightarrow \bigwedge_i \psi_i)$, along the cut when Sys^x is, in fact, defined we obtain (modulo failures)

$$IsSysView(x) \Rightarrow \bigwedge_p (\mathsf{ver}(p) = x) \Rightarrow \bigwedge_p (K_p \Diamond (IsSysView(x)))$$

This is *not* eventual common knowledge [11] of the existence of Sys^x [16]. In essence, our 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. Because a process can never know the composition of $\mathcal{UP}(c)$, it can never know whether the processes in $\mathsf{Memb}_p^x \cap \mathcal{UP}(c)$ have updated their local views to reflect "$!x$" or have crashed. Notice also that GMP is not even required to obtain hindsight about previous system views. This would be phrased as, "at some point in the future, p knows that, at some point in the past, the xth system view existed" :

$$(\mathsf{ver}(p) = x) \Rightarrow \Diamond K_p \Diamond\!\!\!\!- (\mathsf{Memb}_p^x = \mathsf{Sys}^x)$$

though in our protocol, this may be achieved. Upon receipt of the xth commit message, "$!x$", p can reason about the past. It knows that other processes, also in $\mathsf{Memb}(p)$ and still functioning, received and responded to the xth invitation, "$?x$". Because channels are FIFO, p also knows these processes received "$!x-1$". That is, when p receives "$!x$", p knows Sys^{x-1} was a defined system view :

$$(\mathsf{ver}(p) = x) \Rightarrow K_p \Diamond\!\!\!\!- IsSysView(x-1) \tag{4}$$

Equation 4 holds along any consistent cut containing p's receipt of "$!x$". Notice, though, that it is only the existence of successive views that give a process deeper knowledge of past views.

Since $IsSysView(x) \Rightarrow \bigwedge_p \mathsf{ver}(p) = x$, we obtain

$$(\mathsf{ver}(p) = x) \Rightarrow K_p \Diamond\!\!\!\!- \bigwedge_q (\mathsf{ver}(q) = x-1) \Rightarrow K_p \Diamond\!\!\!\!- \bigwedge_q K_q \Diamond\!\!\!\!- IsSysView(x-2) \Leftrightarrow K_p \Diamond\!\!\!\!- E \Diamond\!\!\!\!- IsSysView(x-2)$$

Conjoining over all processes, p, when Sys^x is defined (along c_x) we obtain

$$IsSysView(x) \Rightarrow \bigwedge_p (\mathsf{ver}(p) = x) \Rightarrow \bigwedge_p K_p \Diamond\!\!\!\!- (E \Diamond\!\!\!\!- IsSysView(x-2)) \Leftrightarrow E \Diamond\!\!\!\!- (E \Diamond\!\!\!\!- IsSysView(x-2))$$

[16] Eventual common knowledge would be $\bigwedge_p \Diamond K_p IsSysView(x)$.

That is, processes only have knowledge about each others' local views after the fact. Unwinding the above equations gives the general result

$$IsSysView(x) \Rightarrow (E\Diamond)^y(IsSysView(x-y)).$$

When we assume **Mgr** does not fail, we obtain a higher level of consensus than our specification requires. When p receives "$!x$", it knows that there is some consistent cut *that includes its current, local state* (*i.e.* p does not take any further steps) along which every other functional process in the group will also receive "$!x$". That is, p knows that 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 [17], whose formal semantics are beyond the scope of this paper. This operator is exactly what differentiates concurrent knowledge from other epistemic formulations (for example [11]).

Then the above statements give

$$(\mathsf{ver}(p) = x) \Rightarrow K_p P_p(IsSysView(x)).$$

Finally, letting $\mathcal{G}_c = \mathsf{Sys}^x \cap \mathcal{UP}(c)$, the following holds along any cut where Sys^x is defined :

$$IsSysView(x) \Rightarrow (\bigwedge_{p\in\mathcal{G}_c}(\mathsf{ver}(p) = x)) \Rightarrow \bigwedge_{p\in\mathcal{G}_c}(K_p P_p IsSysView(x)) \equiv$$

$$(IsSysView(x) \Rightarrow E^C_{\mathcal{G}_c}(IsSysView(x))) \tag{5}$$

Equation 5 is the induction rule for concurrent common knowledge; thus, the composition and existence of the xth system view are concurrent common knowledge. Alternatively, in the terminology of [21], c_x is a *locally-distinguishable consistent cut*, also sufficient for concurrent common knowledge.

This is not the case when **Mgr** can fail. When p receives "$!x$", from either **Mgr** or a reconfigurer, it does not know whether the broadcaster failed before completing the broadcast. If so, then p will have to be part of a (further) reconfiguration attempt. The GMP specification only guarantees p that *eventually* Sys^x will be defined :

$$((\mathsf{ver}(p) = x) \Rightarrow K_p\Diamond(IsSysView(x))) \Rightarrow$$

$$((\mathsf{ver}(p) = x) \Rightarrow K_p\Diamond(\bigwedge_{q\in\mathcal{G}_c}(\mathsf{ver}(q) = x))) \Rightarrow$$

$$(\mathsf{ver}(p) = x) \Rightarrow K_p\Diamond(\bigwedge_{q\in\mathcal{G}_c}(K_q\Diamond(\bigwedge_{q'\in\mathcal{G}_c}(\mathsf{ver}(q') = x)))) \cdots$$