Practical Utility of Knowledge-Based Analyses: Optimizations and Optimality for an Implementation of Asynchronous, Fail-Stop Processes (Extended Abstract)

Aleta M. Ricciardi*

TR 91-1233
September 1991

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

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

Practical Utility of Knowledge-Based Analyses : Optimizations and Optimality for an Implementation of Asynchronous, Fail-Stop Processes[1] (Extended Abstract)

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

September 13,1991

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

Abstract

The *Group Membership Problem* is concerned with propagating changes in the membership of a group of processes to the members of that group. A restricted version of this problem allows one to implement a fail-stop failure model of processes in an asynchronous environment assuming a crash failure model. While the ISIS Toolkit relies on this for its Failure Detector, the current specification of GMP sheds no light on how to implement it. We present a knowledge-based formulation, cast as a commit-style problem, that is not only easier to understand, but also makes clear where optimizations to the ISIS implementation are and are not possible. In addition, the epistemic formulation allows us to use the elegant results of knowledge-acquisition theory to discover a lower bound on the required number of messages, construct a minimal protocol, and discuss the tradeoffs between the message-minimal protocol and the optimized ISIS implementation.

1 Introduction

Process groups have found widespread use in distributed systems; they arise whenever processes cooperate to perform a task, provide replication for fault-tolerance, *etc.*. Processes join a group when they recover or desire to participate in the group's activity, and leave a group when they fail. The general *Group Membership Problem* deals with propagating a sequence of changes in a process group's composition to the members of that group. In [14], Ricciardi and Birman defined a particular version of GMP, and used it to implement fail-stop processes in an asynchronous environment assuming a crash failure model. In reality, this 'failure detector' is central to the Isis Toolkit ([2], [3]) so both correctness and speed cannot be undervalued. Unfortunately, the original specification of GMP [14], while technically sound, sheds no light on how to construct a correct, fast solution; it is, instead, a description of an execution's desired observable behavior.

In this abstract, we reformulate GMP as a knowledge-based, commit-style problem, viewing any solution to GMP as one of acquiring and propagating knowledge [9]. This approach proved superior to the behavioral with respect to implementing the failure detector. We show here how the epistemic formulation makes clear that GMP requires majority corroboration, and where optimizations to the GMP solution in [14] 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 GMP, construct a message-minimal solution, and finally quantify the tradeoff 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 ([7], [6], [11], [12], [8]). Our work 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.

GMP is easily stated as a commit problem. While most often associated with database contexts [1], a commit problem is one in which a unique action must be taken by a group of processes. When the group takes this action, the action is said to be *committed* and is (often) irrevocable. In a *centralized* commit protocol, a distinguished process is responsible for coordinating the commit among the set of *outer* processes. If it is known *a priori* that the distinguished process (hereafter denoted *mgr*) will never fail then a reliable broadcast suffices to propagate and coordinate the action, but if *mgr* can fail, more complex protocols are required. Among the issues that must be addressed upon *mgr*'s failure are determining a new *mgr*, and re-establishing local commit consistency.

GMP differs from the Atomic Commit Problem considered in [6] and the Negotiated Commit Problem [12] in a number of ways. Most notably, GMP permits processes to vote for more than one value over time. As voting is not stable, committing a value becomes more complex, since the conditions validating the commit may no longer exist, and especially when the coordinator fails. We also use a modified notion of 'correct' process, that accounts for the indistinguishability of process failures and communication asynchrony.

Section 2 describes the environment and model of computation, and Section 3 briefly discusses the formal logic. Section 4 presents the Strict Group Membership Problem, and uses the logic to specify it formally as a commit problem. Section 5 contains the optimizations and optimality proofs.

2 The Environment and Model

We consider a distributed system in which processes communicate only by passing messages to each other, and in which both processes and communication channels are *asynchronous*. The communication network is assumed completely-connected and point-to-point, and its channels are assumed reliable (eventual, exactly-once delivery of uncorrupted messages) and FIFO. Processes fail by crashing, but due to communication asynchrony, such events are impossible to detect accurately. Nonetheless, we speculate that there is some means by which a process comes to *suspect* another one faulty, and require it to *gossip* this belief to all others with which it can communicate. We further require that a process never receive a message from one it believes faulty, thereby *isolating* the believed-faulty process.

Denote by Proc a finite set of process identifiers, $\{p_1, \ldots, p_n\}$. A *history* for process p, h_p, is a sequence of events executed by p, and must begin with the distinct event $start_p$. Processes may send and receive messages, and do internal computation. The event $send_p(q,m)$ denotes p sending message m to q, and $recv_q(p,m)$ denotes q's receipt of m from p. The distinct event $quit_p$ models the crash failure of process p, after which only other $quit_p$ are permitted. Process p executes $faulty_p(q)$ upon suspecting q to be faulty or receiving a message gossiping q's faultiness.

A *cut* is an n-tuple of process histories, one for each process in Proc, $c = (h_{p_1}, h_{p_2}, \ldots, h_{p_n})$. We assume familiarity with Lamport's *happens-before* relation and *consistent cuts*[10].

The indexical set $\mathsf{Up}(c)$ is the subset of Proc whose members are functional along consistent cut c.

3 The Logic

Our specification language is a blend of (branching time) tense and epistemic [9] [5] logics. The basic semantic entities of this logic are consistent cuts; *i.e.* logical formulas are evaluated along consistent cuts, as in [15] and [13]. Informally, two cuts are p-equivalent exactly when the local state of p in each cut is identical; they are causally related if one is a prefix of the other. The following modalities are used :

- $K_p\phi$ - "p knows ϕ". Holds along c when ϕ holds along every p-equivalent cut.
- $I_G\phi$ - "the group G implicitly knows ϕ". Holds along c when some member of G knows ϕ.
- $\Box\phi$ - "ϕ is henceforth true". Holds along c when every cut containing c as a prefix satisfies ϕ.
- $\boxminus\phi$ - "ϕ has always been true". Holds along c when every prefix of c satisfies ϕ.

The indistinguishability of failures from communication delays seems to warrant using the deontic modality [16] 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 deontic interpretation would give equal weight to belief in faultiness and belief in non-faultiness, our model favors one before the event $faulty_p(q)$ 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. In fact, we have modeled *fail-stop* process failures.

We express this ambiguity with two formulas, one of which is *local* to the suspecting process while the other is local to no process. Formula ϕ is local to p if p always knows whether it is true [5] : $K_p\phi \vee K_p\neg\phi$ holds along all consistent cuts. In this way, the standard knowledge operator models exactly the behavior process p exhibits upon executing $faulty_p(q)$. Throughout this abstract, the statement "p believes (or knows) q is faulty" should be taken as an artifact of our model's behavioral requirements, not literally.

Chandra and Toueg impart a pure deontic interpretation to local failure beliefs [4]. 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.

4 The Group Membership Problem

The general *Group Membership Problem* is concerned with propagating changes in a process group's composition to each of its members. In many situations, a group functions correctly only when its members have identical local views of its composition. For example, in a token passing implementation of atomic broadcast, processes' local views and a static, linear ordering on process identifiers determine which process each group member believes holds the token. Thus, an important instance of GMP ensures that each member of a group sees identical changes to the group's composition, and in the same order. Other pertinent issues include partitions, and join and leave behavior. This abstract is concerned with a single process group and the particular instance of GMP described in [14], hereafter Strict GMP.

Let $\mathsf{Memb}_p(c)$ denote p's local view of the group along consistent cut c, and let Memb_p^x denote the x^{th} instance of p's local view. The *system view* determined by S along c is defined to be :

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

That is, if no members of S are functional, the system view is empty; if all functional members have identical local views, the system view is any local view; and if the functional members' local views disagree, the system view is undefined.

In Strict GMP, every system execution must exhibit a maximal sequence of *temporally unique* (at most one system view exists along any cut) system views. Moreover every member of a system view must also be a member of the group determining it. Since Strict GMP requires a sequence of system views, integer instances are well-defined; let Sys^x be the x^{th} *version* of the system view. We assume a commonly-known, linear rank on the members of Sys^x , in practice determined by length of membership.

The Appendix contains a skeleton of the solution to Strict GMP presented in [14]. This protocol (hereafter AFS for Asynchronous Fail-Stop) is a combination of two- and three-phase centralized commit protocols[1]. If *mgr* is not believed faulty for 'suitably long'[2] a two-phase commit protocol suffices. If *mgr* is ever believed faulty, a three-phase *reconfiguration* protocol is run. A process initiates reconfiguration if it believes every process of higher rank than itself is faulty. In the first phase, the initiator collects local view information from the outer processes, and determines an update that would re-establish local view consistency. In the second phase, it proposes this update and awaits responses, broadcasting a commit in the third phase.

AFS is a *full-information protocol*; all relevant state information is sent with each protocol message. In Strict 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.

[1] A communication phase consists of a process *broadcasting* a message to a group of processes, and collecting their responses to it. In truth, this protocol is one-and-one-half (broadcast, collection, broadcast), and two-and-one-half phase protocols, but this is awkward.

[2] for the duration of an update it initiates

For the rest of this abstract, the phrase "submission for version x" refers to either the value *mgr* invites processes to commit for version x, or to a reconfigurer's proposal for version x.

4.1 Strict GMP Definitions

Strict GMP, as a commit protocol, requires functional processes in a given system 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 $(x-1)^{st}$ system view must commit to install the x^{th} system view : $V \subseteq \{\text{remove}, \text{add}\} \times \mathsf{Proc}$. Before committing a given update, a process may *vote* for more than one update value. This differs from the Atomic Commit specification of [6], as we must model the occurrence of conflicting proposals from competing coordinators. 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 system view for which v is submitted and/or committed. The same value may be submitted for different versions.

The following are formulas, events, and notation used to rephrase Strict GMP as a commit problem. Note that events are written in *italics*, while formulas are in *slanted* type.

- $\mathit{vote}_p(v_x)$ holds along consistent cut c if $\mathit{vote}_p(v_x)$ is the most-recent voting event p, with local view Memb_p^{x-1}, executed.
- $\mathit{commit}_p(v_x)$ holds if p executed the commit event $\mathit{commit}_p(v_x)$ to form its x^{th} local view.
- $\mathit{faulty}_p(q)$ holds if p has executed $\mathit{faulty}_p(q)$. We omit the subscript when we are not concerned with which process believes q faulty.

  The formulas $\mathit{faulty}_p(q)$, $\mathit{vote}_p(v_x)$, and $\mathit{commit}_p(v_x)$ are local to p.
- $\mathit{down}(q)$ holds if q has crashed. The distinction between $\mathit{down}(q)$ and $\mathit{faulty}_p(q)$ is that $\mathit{down}(q)$ is never local to any process : $(\Box\neg)K_p \mathit{down}(q) \wedge (\Box\neg)K_p\neg \mathit{down}(q)$.
- $\mathsf{Maj}(S)$ is the set of all majority subsets of S.
- $\mathit{StableVote}(v_x)$ holds when there is a subset of Sys^{x-1} , each process of which is either crashed or has most-recently voted for v_x and will not vote for another v'_x. Moreover, every process not in this subset is believed faulty by some process in it :

$$\mathit{StableVote}(v_x) \stackrel{\text{def}}{=} \bigvee_{S \subseteq \mathsf{Sys}^{x-1}} \left(\bigwedge_{p \in S} \left(\mathit{vote}_p(v_x) \wedge \bigwedge_{v' \neq v} \Box\neg \mathit{vote}_p(v'_x) \right) \wedge \bigwedge_{q \notin S} I_S \mathit{faulty}(q) \right)$$

  We use $\mathit{StableVotes}_S(v_x)$ when referring to a particular set, S, according to which v_x is stable.

4.2 Strict GMP Specification

A protocol is a solution for Strict GMP if every execution of it satisfies [3] :

[3] In fact, Strict GMP has other requirements (*e.g.* constraining permissible votes, and specifying initial conditions) but these are not relevant to the current work.

Validity If p commits v_x, p is in a subset of Sys^{x-1} according to which v is stable for x :

$$commit_p(v_x) \Rightarrow \bigvee_{S \subseteq \mathsf{Sys}^{x-1}} \Big(p \in S\Big) \wedge StableVotes_S(v_x)$$

Uniqueness If p commits v_x, no other process, q, ever commits v'_x, for $v' \neq v$:

$$commit_p(v_x) \Rightarrow \bigwedge_{v' \neq v} \bigwedge_{q \in \mathsf{Proc}} \Box \neg commit_q(v'_x)$$

Totality If p commits v_x, then for every other process, q, either **1)** q is not in Sys^x or **2)** q eventually commits v_x or **3)** q eventually fails :

$$commit_p(v_x) \Rightarrow \bigwedge_{q \in \mathsf{Proc}} \Bigg(\Big(q \notin \mathsf{Sys}^x\Big) \vee \Big(\Diamond commit_q(v_x)\Big) \vee \Big(\Diamond down(q)\Big)\Bigg)$$

It is easy to see [14] that Uniqueness cannot be guaranteed without additional restrictions on stability.

Definition Value v_x is *committably stable* (c-stable) if and only if $\exists S \in \mathsf{Maj}(\mathsf{Sys}^{x-1}).\Big(StableVotes_S(v_x)\Big)$. The formula $C\text{-}Stable(v_x)$ holds exactly when v_x is c-stable ■

Uniqueness and Validity then combine to restate the latter as

$$commit_p(v_x) \Rightarrow \bigvee_{G \in \mathsf{Maj}(\mathsf{Sys}^{x-1})} \Big(p \in G\Big) \wedge C\text{-}Stable_G(v_x).$$

5 Optimality

These analyses use two notions of optimality : 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-optimal* if it is impossible to commit a value safely in fewer messages. In this section, we show that parts of the AFS protocol are knowledge-minimal, and how the knowledge-based formulation of Strict GMP led to optimizations in the other three parts. We also derive a lower bound on the number of messages required to solve Strict GMP, and compare AFS with a message-optimal protocol. While we believe the optimizations to be at least knowledge-minimal, different failure scenarios make analysis of whether they are optimal in either sense beyond the scope of this work.

For these purposes, the most important aspect of correctness (proven in [14]) is that at most one value attains c-stability for any given version. Thus, a process may *safely* commit v_x as soons as v_x becomes c-stable. On the other hand, a simple derivation from Validity shows that every solution to Strict GMP necessarily satisfies $commit_p(v_x) \Rightarrow K_p C\text{-}Stable(v_x)$, so the 'earliest' a process can commit v_x is as soon as it knows v_x is c-stable : $K_p C\text{-}Stable(v_x) \Rightarrow commit_p(v_x)$. We call such a commit protocol a $1K$-commit protocol since a process can commit with one 'level' of knowledge.

Denote by $\texttt{M-sub}(v_x)$ *mgr* 's Phase I invitation message, and by $\texttt{M-com}(v_x)$ its Phase II commit message. Let $\texttt{R-int}(x)$ denote the reconfigurer's Phase I interrogation query when in local version $x-1$, $\texttt{R-sub}(v_x)$ its Phase II proposal, and $\texttt{R-com}(v_x)$ its Phase III commit message. We use $\texttt{com}(v_x)$ be any commit message ($\texttt{M-com}(v_x)$ or $\texttt{R-com}(v_x)$), and $\texttt{sub}(v_x)$ for any submit message.

Definition Let p' send message m, and let $\mathbf{ack}(m)$ denote a message sent by a recipient of m, back to p' acknowledging receipt of m.

- $\mathsf{Recipients}(p', m) \stackrel{\text{def}}{=} \left\{p \mid recv_p(p',\mathrm{m})\right\}$
- $\mathsf{AcksSent}(p', m) \stackrel{\text{def}}{=} \left\{p \mid send_p(p',\mathbf{ack}(m))\right\}$
- $\mathsf{AcksRcvd}(p', m) \stackrel{\text{def}}{=} \left\{p \mid recv_{p'}(p,\mathbf{ack}(m))\right\}$ ∎

Observe that, for any p' and m, $\mathsf{AcksRcvd}(p', m) \subseteq \mathsf{AcksSent}(p', m) \subseteq \mathsf{Recipients}(p', m)$. This is significant in the next definition, and in the propositions that follow. To describe the most general, observable system state from which it is inferable that a value is c-stable, we define a 'successful initiator' to be one whose submission can possibly be committed.

Definition Process p' is *successful for v_x* if and only if a majority subset of Sys^{x-1} acknowledge p''s submission : $\mathsf{AcksSent}(p', \mathbf{sub}(v_x)) \in \mathsf{Maj}(\mathsf{Sys}^{x-1})$. ∎

This definition leaves open whether p' actually received all the acknowledgements sent to it. While this is obviously relevant in determining whether p' is able to commit the update, in the absence of concrete evidence, it is impossible for a subsequent reconfigurer to *know* whether p' succeeded in committing the update anywhere. However, Uniqueness and Totality require a reconfigurer to assume the update was committed ('invisibly' to it) if it determines that p' could *possibly* have issued the commit message ([14] covers this issue in more detail).

Correctness of the AFS protocol implies :

Fact 5.1 *In* AFS *if p' is successful for v_x, no value unequal to v is thereafter submitted for version x.*

Fact 5.2 *In* AFS *if p' is successful for v_x, then v_x is c-stable.*

Fact 5.3 *In* AFS *if* $\mathrm{vote}_p(v_x)$ *holds for a majority of* Sys^{x-1} *along any consistent cut, then C-Stable(v_x) holds along that cut :* $\left(\bigvee_{G\in \mathsf{Maj}(\mathsf{Sys}^{x-1})}\left(\bigwedge_{p\in G} \mathrm{vote}_p(v_x)\right)\right) \Rightarrow \textit{C-Stable}(v_x)$.

5.1 Knowing Stability

To understand when a process executing AFS knows a value is c-stable, we analyze the protocol's communication phases. We show two communication phases are necessary when the initiator is *mgr*. In the interest of brevity, we restrict this analysis to instances of AFS when a given process is either the *first mgr*, or has been *mgr* for at least one completed update of the system view; the issues arising in the transition of a process from reconfiguration initiator to *mgr* are too complex for discussion here.

We also show that in three of the four 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 AFS , allowing us, in two cases, to eliminate a full phase of communication, and to preclude a third entirely. We show it is impossible to improve the fourth.

Definition Let *IsMgr*(p', x) hold if a majority of Sys^{x-1} believe p' is the highest-ranked, non-faulty process. Then version x has a *clean starting point* if $x = 1$, or *IsMgr*$(p', x-1)$ held throughout the formation of Sys^{x-1} . ∎

Fact 5.4 *In* AFS *if x has a clean starting point, and if mgr submits v_x, then no process in* Sys^{x-1} *has previously voted for v_x.*

Proposition 5.1 *In* AFS K_{mgr} *C-Stable*$(v_x) \Leftrightarrow$ $\mathsf{AcksRcvd}(mgr, \texttt{M-sub}(v_x)) \in \mathsf{Maj}(\mathsf{Sys}^{x-1})$.

Proof "$\Rightarrow$" We first show $K_{mgr}\ vote_p(v_x) \Rightarrow p \in \mathsf{AcksRcvd}(mgr, \texttt{M-sub}(v_x))$.

Building on [5], Mazer [12] showed that if p learns ϕ_q, a formula local to q, and if processes **a)** are asynchronous, or **b)** can experience crash failures and ϕ_q can only be made true by q, then p must receive a message from, or indirectly from, q implying ϕ_q. $vote_q(v_x)$ 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 Sys^{x-1} had voted for v_x before receiving $\texttt{M-sub}(v_x)$ (Fact 5.4). Inspecting AFS shows that a process votes after receiving $\texttt{M-sub}(v_x)$ and before responding to it, and since $vote_p(v_x)$ has not held previously, it holds for the first time in the execution immediately before p acknowledges $\texttt{M-sub}(v_x)$. *mgr* can infer $vote_p(v_x)$ upon receipt of p's response to $\texttt{M-sub}(v_x)$. Finally, outer processes do not send messages to one another[4], so no other process can have learned $vote_p(v_x)$ independently. As a result, *mgr* cannot have learned $vote_p(v_x)$ from a process other than p. The only message p sends is $\textbf{ack}(\texttt{M-sub}(v_x))$ so $K_{mgr}\ vote_p(v_x) \Rightarrow p \in \mathsf{AcksRcvd}(mgr, \texttt{M-sub}(v_x))$.

Let $vote_G(v_x)$ denote $\bigwedge_{p \in G} vote_p(v_x)$. Then,

$$K_{mgr}\ \textit{C-Stable}(v_x) \Rightarrow \bigvee_{G \in \mathsf{Maj}(\mathsf{Sys}^{x-1})} K_{mgr}\ \textit{C-Stable}_G(v_x) \Rightarrow K_{mgr}\ vote_G(v_x) \Rightarrow$$
$$G \subseteq \mathsf{AcksRcvd}(mgr, \texttt{M-sub}(v_x)) \Rightarrow \mathsf{AcksRcvd}(mgr, \texttt{M-sub}(v_x)) \in \mathsf{Maj}(\mathsf{Sys}^{x-1}).$$

"$\Leftarrow$" The composition of $\mathsf{AcksRcvd}(mgr, \texttt{M-sub}(v_x))$ is local to *mgr*. Since $\mathsf{AcksRcvd}(mgr, \texttt{M-sub}(v_x)) \subseteq \mathsf{AcksSent}(mgr, \texttt{M-sub}(v_x))$, *mgr* knows it is successful for v_x. Using Fact 5.2, K_{mgr} *C-Stable*(v_x). ∎

Proposition 5.1 shows that whenever x has a clean starting point *mgr* cannot know c-stability of any value until after collecting responses. We now use this to show that AFS , during a *mgr*-initiated update begun from a clean starting point, is knowledge-minimal.

Proposition 5.2 (Two-Phase Necessity) *In* AFS *if x has a clean starting point, then a process commits v_x during a mgr -initiated update as soon as it knows C-Stable(v_x) :*

1. $send_p(mgr, \textbf{ack}(\texttt{M-sub}(v_x))) \Rightarrow \boxminus \neg K_p\, \textit{C-Stable}(v_x)$
2. $recv_p(mgr, \texttt{M-com}(v_x)) \Rightarrow K_p\, \textit{C-Stable}(v_x)$

Proof Since x has a clean starting point, no process voted for v_x before receiving $\texttt{M-sub}(v_x)$. Proposition 5.1 and inspecting the protocol show that *mgr* does not know *C-Stable*(v_x) until after it has sent all $\texttt{M-sub}(v_x)$ messages; therefore, $\texttt{M-sub}(v_x)$ cannot have implied *C-Stable*(v_x). Furthermore, neither p's internal voting event nor its acknowledgement add to its knowledge [5], establishing *(1)*. Outer processes do not communicate amongst themselves so p cannot have learned *C-Stable*(v_x) independently or from another outer process since receiving $\texttt{M-sub}(v_x)$. In consequence, the earliest p can learn *C-Stable*(v_x) is upon receipt of *mgr* 's second phase (commit) message. Since AFS is full-information, *C-Stable*(v_x) is propagated by $\texttt{M-com}(v_x)$. ∎

5.2 Optimizations

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

[4] except during reconfiguration, in which case *mgr* has been isolated

Fact 5.5 *Let r be a reconfiguration initiator with local version $x-1$. Then in* AFS *, when collecting interrogation responses exactly one of the following three scenarios is possible :* **a)** *r learns some process has local version x;* **b)** *r learns some process has local version $x-2$;* **c)** *all processes from which r receives responses have local version $x-1$.*

In the first instance, r learns *C-Stable*(v_x) at the end of reconfiguration Phase I. To optimize AFS , r, instead of proposing v_x and collecting responses, commits v_x and propagates *C-Stable*(v_x) to any process whose acknowledgement also indicated local version $x-1$. This optimization saves approximately $2|\mathsf{Sys}^{x-1}|$ messages. Another optimization precludes case **b)**, for upon receiving $\texttt{R-int}(x)$, a process with local version $x-2$ learns *C-Stable*(v_{x-1}). It can commit that update and respond to the reconfigurer with its new version. In **c)**, if a majority of Sys^{x-1} (among the interrogation respondents) also indicate having most-recently voted for the same value, r learns that value is c-stable; as in the first optimization, reconfiguration Phase II is unnecessary.

Proposition 5.3 (Optimization 1) *If r is a reconfiguration initiator with local version $x-1$, and some process's response to* $\texttt{R-int}(x)$ *indicates local version x, then r learns C-Stable(v_x) upon receipt of* $\texttt{ack}(\texttt{R-int}(x))$ *from that process.*

Proof Upon receipt of $\texttt{ack}(\texttt{R-int}(x))$ from any p with local version x, $K_r commit_p(v_x)$. Distributing K_r over implication in Validity gives $K_r C\text{-}Stable(v_x)$. ∎

Proposition 5.4 (Optimization 2) *If r is a reconfiguration initiator with local version $x-1$, then any process, p, with local version $x-2$ learns C-Stable(v_{x-1}) upon receipt of* $\texttt{R-int}(x-1)$ *from r.*

Proof Similar to Proposition 5.3 ∎

Proposition 5.5 (Optimization 3) *Let r be a reconfiguration initiator with local version $x-1$, and suppose every process in* $\mathsf{AcksRcvd}(r, \texttt{R-int}(x))$ *indicates local version $x-1$. If a majority of* Sys^{x-1} *(among* $\mathsf{AcksRcvd}(r, \texttt{R-int}(x))$*) also indicate having most-recently voted for some value, v_x , then $K_r C\text{-}Stable(v_x)$.*

Proof Follows from Fact 5.3. ∎

The final proposition shows AFS is knowledge-minimal when all respondents to $\texttt{R-int}(x)$ report the same local version, but no majority subset has most-recently voted for the same value.

Proposition 5.6 (Three-Phase Necessity) *Let r be a reconfiguration initiator with local version $x-1$, and suppose every process in* $\mathsf{AcksRcvd}(r, \texttt{R-int}(x))$ *indicates local version $x-1$. If no majority subset of* Sys^{x-1} *(among* $\mathsf{AcksRcvd}(r, \texttt{R-int}(x))$*) indicate they have most-recently voted for the same value, then for all v_x and $p \in$* $\mathsf{AcksRcvd}(r, \texttt{R-int}(x))$,

1. $send_p(r, \texttt{ack}(\texttt{R-sub}(v_x))) \Rightarrow \boxminus \neg K_p C\text{-}Stable(v_x)$

2. $recv_p(r, \texttt{R-com}(v_x)) \Rightarrow K_p C\text{-}Stable(v_x)$

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

Given that no majority of r's respondents have 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; it may be able to envision scenarios in which each of the reported values are c-stable. Correctness of AFS only ensures that if a value *is* c-stable, r will propose that value; if v'_x and v_x are values r's respondents report pending, correctness of AFS only guarantees $send_r(p,\texttt{R-sub}(v_x)) \Rightarrow K_r \neg C\text{-}Stable(v'_x)$, and implies nothing about whether v_x is c-stable. Thus, $\texttt{R-sub}(v_x)$ does not imply $C\text{-}Stable(v_x)$, and since outer processes have not sent messages amongst themselves, none can have learned c-stability (before receiving $\texttt{R-sub}(v_x)$) independently. Again, neither p's vote nor its response add to its knowledge.

As a side note, r choosing to propose v_x and an outer process receiving this proposal does not even guarantee v_x will become c-stable. This is due to possible failures in the second and third phases of reconfiguration : r may fail before sending all the proposal messages, or a majority may fail before receiving and/or voting for the proposal, both of which result in v_x not becoming c-stable. Moreover, both r and the outer processes can envision these scenarios.

As in Proposition 5.2, r learns $C\text{-}Stable(v_x)$ if $\mathsf{AcksRcvd}(r, \texttt{R-sub}(x_x))$ is a majority subset of Sys^{x-1} , and outer processes learn it upon receipt of r's commit message. ∎

5.3 Minimality

We use Mazer's Message Chain Theorem [12] to determine the minimal number of messages required by any solution to Strict GMP. Similar work for different commit-style problems is in [6] and [11]. For simplicity, we assume no process has knowledge of another's votes.

Let $\mu_S = \left\lfloor \frac{|S|}{2} \right\rfloor + 1$. We show that, for a given set S, at least $2(\mu_S - 1)$ messages are necessary for any member of S to learn c-stability of a value, and that any algorithm using fewer than $(|S| - 1) + 2(\mu_S - 1)$ messages is not Total.

We have already shown that a process can commit a value as soon as it knows it to be c-stable, and that any solution to Strict GMP is necessarily a $1K$-commit protocol. Let $G \in \mathsf{Maj}(S)$ and let $C\text{-}Stable_G(v_x)$ hold. Then $K_{p'} C\text{-}Stable(v_x)$ if and only if

$$\bigwedge_{p\in G} \left(K_{p'}\, vote_p(v_x)\right) \wedge \bigwedge_{p\in G} \bigwedge_{v'\neq v} K_{p'} \left(\Box\neg vote_p(v'_x)\right) \wedge \bigwedge_{q\in \overline{G}} K_{p'} I_G faulty(q)$$

Since $vote_p(v_x)$ is local to p, p' must receive (at least) one message for $K_{p'}\, vote_p(v_x)$ to hold, so at least $\mu_S - 1$ messages are needed to satisfy the first conjunct. For the second conjunct, nothing in the specification of Strict GMP ensures p' that p'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[5]. 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 (*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 $\mu_S - 1$ more messages, giving a total of $2(\mu_S - 1)$ messages before p' can commit.

There are two possible patterns of communication :

- 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

[5] This is not true for $|V| \leq 2$.

since it does not yet know whether each outer process has stabilized[6]. This requires an additional $\mu_S - 1$ messages, totaling $3(\mu_S - 1)$ 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 its vote back to p'. This approach requires a minimum of $2(\mu_S - 1)$ messages for p' to learn c-stability.

Finally, Totality 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 $(|\mathsf{S}| - f) \times 2(\mu_S - 1)$ 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 $|\mathsf{S}| - 1$ messages are needed[7], giving a minimal total of $(|\mathsf{S}| - 1) + 2(\mu_S - 1)$ messages.

In AFS , installing version x from a clean starting point and assuming *mgr* is not thought faulty, uses at most $3(|\mathsf{Sys}^{x-1}| - 1) - f$ messages. The extra messages (approximately $|\mathsf{Sys}^{x-1}| - f - 1$) represent the tradeoff between compressing *mgr* 's faultiness determinations to one phase, and spacing them out over possibly $\mu_{\mathsf{Sys}^{x-1}} + 1$ waiting periods[8].

6 Conclusion

This abstract presented a knowledge-based formulation of Strict GMP and an analysis of a particular solution to it. The knowledge-theoretic approach easily identified three ways to optimize AFS and proved the remaining cases knowledge-minimal. We also quantified the minimal number of messages needed to solve Strict GMP under certain assumptions, outlined a minimal algorithm, and compared it with AFS . Given that AFS actually implements a failure detection service for asynchronous systems, its speed and correctness are crucial. The simplicity with which optimizations were identified demonstrates the practical utility of knowledge-based reasoning.

Acknowledgements

The author thanks Ken Birman and Keith Marzullo for their helpful discussions and enthusiasm for this work, and Vassos Hadzilacos for his inspiring lectures on knowledge-based reasoning.

References

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

[2] 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.

[3] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. Technical Report TR-91-1216, Cornell University, July 1991.

[6] A technicality now, but important in practice when reconfiguration may be ongoing.

[7] The lack of f here arises from the impossibility of p' knowing whether another process is truly crashed.

[8] The worst case 'time' occurs when p' chooses the worst sequence of processes from which to get responses : if, from the initial majority subset it chooses, p' receives only $\mu_{\mathsf{Sys}^{x-1}} - 2$ responses, then observes $\left\lceil \frac{|\mathsf{Sys}^{x-1}|}{2} \right\rceil - 2$ successive 'failures' before getting the last response.

[4] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Ashynchronous Systems. In *Proceedings of the Tenth Annual A.C.M. Symposium on Principles of Distributed Computing.* ACM, August 1991.

[5] K. M. Chandy and J. Misra. How Processes Learn. *Distributed Computing*, 1(1):40–52, 1986.

[6] V. Hadzilacos. A Knowledge Theoretic Analysis of Atomic Commitment. Private Communiation, 1991.

[7] J. Y. Halpern. Using Reasoning About Knowledge to Analyze Distributed Systems. *Annaul Review of Computer Science, II*, pages 37–68, 1987. Ed. J.F.Traub, Annual Reviews, Inc.

[8] J. Y. Halpern and R. Fagin. A Formal Model of Knowledge, Action and Communication in Distributed Systems. In *Proceedings of the 4th ACM Symposium on the Principles of Distributed Computing*, pages 224–236, August 1985.

[9] J.Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *JACM*, 1990.

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

[11] M. Mazer and F.H.Lochovsky. Analyzing Distributed Commitment by Reasoning About Knowledge. Technical Report CRL 90/10, Digital Equipment Cambridge Research Lab, 1990.

[12] M. S. Mazer. A Link Between Knowldege and Communication in Faulty Distributed Systems. In R. Parikh, editor, *Proceedings of the 3rd Conference on the Theoretical Aspects of Reasoning About Knowledge*, pages 289–304, 1990.

[13] A. Ricciardi. Completeness of a Temporal Logic for Asynchronous Systems. Technical Report 89-1052, Cornell University Computer Science Department, November 1989.

[14] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Procedings of the Tenth Annual A.C.M. Symposium on Principles of Distributed Computing.* A.C.M., August 19-21 1991. This is an extended abstract of Cornell University Technical Report TR91-1188, of the same name.

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

[16] M. R. Tuttle. *Knowledge and Distributed Computation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1989. Department of Electrical Engineering and Computer Science.

Appendix - The AFS Solution

This is a scaled-down version of the AFS solution. It focuses only on the communication structure, and vote and commit events, while ignoring failure beliefs and events. Let $Bcast_p(G, m)$ denote the set of events $\forall q \in G(send_p(q,m))$. This sequence of events is executed atomically but is not failure-atomic.

Task : Update Algorithm - mgr
/* Local version $x - 1$ */
Begin :
choose an update value v_x ;
$Bcast_{mgr}$ ($\mathsf{Memb}_{mgr}^{x-1}$, M-sub(v_x));
$vote_{mgr}$ (v_x);
$\forall p \in \mathsf{Memb}_{mgr}^{x-1}$ **do**
    $recv_{mgr}$ (p,ack(M-sub(v_x))) **or** $timeout(p)$;
/* End of Response Collection */
if AcksSent(mgr , M-sub(v_x)) $\in$ Maj(Sys^{x-1})
    then
        $Bcast_{mgr}$ ($\mathsf{Memb}_{mgr}^{x-1}$, M-com(v_x));
        $commit_{mgr}$ (v_x);
End.

Task : Update Algorithm - Outer Process, p
Begin :
$recv_p$(mgr ,M-sub(v_x));
$vote_p$(v_x);
$send_p$(mgr ,ack(M-sub(v_x)));
$recv_p$(mgr ,M-com(v_x));
$commit_p$(v_x);
End.

Task : Reconfiguration - Initiator, r
/* Local version $x - 1$ */
Begin :
$Bcast_r$(Memb_r^{x-1}, R-int(x));
$\forall p \in \mathsf{Memb}_r^{x-1}$ **do**
    $recv_r$(p,ack(R-int(x))) **or** $timeout(p)$;
if AcksSent(r, R-int(x)) $\in$ Maj(Sys^{x-1})
    then
        Determine an update proposal, v_x ;
        $Bcast_r$(Memb_r^{x-1}, R-sub(v_x));
        $vote_r$(v_x);
        $\forall p \in \mathsf{Memb}_r^{x-1}$ **do**
            $recv_r$(p,ack(R-sub(v_x))) **or** $timeout(p)$;
        if AcksSent(r, R-sub(v_x)) $\in$ Maj(Sys^{x-1})
            then
                $Bcast_r$(Memb_r^{x-1}, R-com(v_x));
                $commit_r$(v_x);
                Begin Update Algorithm as mgr;
End.

Task : Reconfiguration - Outer Process, p
Begin :
$recv_p$(r,R-int(x));
$send_p$(r,local state information) ;
$recv_p$(r,R-sub(v_x));
$vote_p$(v_x);
$send_p$(r,ack(R-sub(v_x)));
$recv_p$(r,R-com(v_x));
$commit_p$(v_x);
Begin Update Algorithm with r as new mgr;
End.