A Secure Group Membership Protocol

Michael K. Reiter

Abstract—A group membership protocol enables processes in a distributed system to agree on a group of processes that are currently operational. Membership protocols are a core component of many distributed systems and have proved to be fundamental for maintaining availability and consistency in distributed applications. In this paper we present a membership protocol for asynchronous distributed systems that tolerates the malicious corruption of group members. Our protocol ensures that correct members control and consistently observe changes to the group membership, provided that in each instance of the group membership, fewer than one-third of the members are corrupted or fail benignly. The protocol has many potential applications in secure systems and, in particular, is a central component of a toolkit for constructing secure and fault-tolerant distributed services that we have implemented.

Index Terms—Security, reliability, distributed systems, group membership protocol, Byzantine failures.

1 Introduction

A *group membership protocol* is a protocol by which processes in a distributed system can reach agreement on a group of processes that are currently operational. A process may need to be removed from the group if the process fails or is perceived to fail because, for instance, it is disconnected from the network. A process may need to be added to the group when, for example, it rejoins the system after recovering from a failure. It is the duty of the membership protocol to ensure that processes observe changes to the group membership in some consistent fashion. Membership protocols have received much attention in the scientific literature (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9]) and have been implemented in numerous experimental and commercial systems (e.g., [10], [1], [11], [12], [7]). They have proved to be fundamental for maintaining consistency and availability despite process failures in a wide range of distributed applications.

In this paper we present a membership protocol that is suitable for use in distributed systems in which some processes may be corrupted by a malicious intruder. More precisely, our protocol provides strong consistency guarantees regarding the manner in which correct processes observe changes to the group membership, despite the efforts of corrupted processes inside or outside the group. Moreover, malicious processes cannot single-handedly effect changes to the group membership or prevent needed changes from occurring. Our protocol achieves these guarantees in an asynchronous system, provided that in each instance of the group membership, fewer than one-third of the group members are corrupted or fail benignly. To differentiate our work from others, we note that our protocol is *not* concerned with the *detection* of corrupt group members (although our protocol can be used to remove them from the group once detected). Thus, its purpose differs from that of intrusion detection systems (e.g., [13]). Moreover, the ability of our protocol to tolerate the corruption of group members sets it apart from previous security work in group-oriented systems that focuses on securing group semantics and communication against attacks from outside the group only (e.g., [14]).

There are many motivations for a secure group membership protocol. First, as membership protocols play important roles for many distributed applications, they may also present avenues through which intruders can mount attacks on the availability and integrity of distributed systems. By manipulating the membership protocol underlying a replicated service, for example, an intruder might effect the removal of sufficiently many servers from the server group to deny service to clients. Similarly, the intruder might cause servers or clients to observe inconsistent group memberships, which could result in inconsistent replies to clients if, say, each reply is computed from the inputs of some fraction of the group members (e.g., [15], [16]). Use of our protocol to maintain membership information would prevent these attacks.

A second motivation for our protocol is that it facilitates the realization of other security technologies. For example, group-oriented cryptographic controls and secure group computing techniques have received substantial attention in the scientific literature (see [17], [18]). These techniques typically require coordination among group members, and therefore, their use in real systems can be facilitated by providing members with consistent group membership information that cannot be manipulated by corrupt members.

An example of this can be found in a proposed technique for using group-oriented cryptographic controls to construct distributed services that remain available and correct despite the corruption of some of their component servers [19]. This technique requires that client requests be issued to servers by an *atomic broadcast* protocol, which ensures that all correct servers receive the same sequence of requests. Assuming that only crash failures occur, several systems provide practical and efficient implementations of atomic broadcast with the

• *M.K. Reiter is with AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: reiter@research.att.com.*

Manuscript received November 1993; revised April 1995.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number S95064.

help of a membership protocol (e.g., [20], [12]). With our membership protocol and similar techniques, we have built an efficient implementation of atomic broadcast that can tolerate even malicious process behavior [21], [22]. Our protocol thus contributes to a set of mechanisms that make the techniques of [19] practical, and we have implemented a toolkit, called Rampart, for building secure, fault-tolerant services using these techniques [22].

The rest of this paper is structured as follows. In Section 2, we describe our assumptions about the system. In Section 3, we more carefully define the properties that our membership protocol satisfies. We give a high-level presentation of our protocol in Section 4, deferring a formal treatment to Appendix A. We discuss performance in Section 5 and conclude in Section 6. We prove the correctness of our protocol in Appendix B.

2 The System Model

We assume a system consisting of some countable, possibly infinite number of *processes* $p_0, p_1, p_2, \ldots$ We will often denote processes with the letters p, q, and r when subscripts are unnecessary. We allow an infinite number of processes to model infinite executions in which processes are continually created. However, at any point in an execution, only a finite number of processes are present. A process that always behaves according to its specification is said to be *correct*. A *faulty* process, however, can behave in any fashion whatsoever (i.e., Byzantine failures), limited only by the assumptions stated below.

Processes communicate exclusively by sending and receiving messages over a completely connected, point-to-point network. Communication channels are authenticated and protect the integrity of communication using, e.g., well-known cryptographic techniques [23]. Communication is reliable but asynchronous: If the sender and destination of a message are correct, then the destination will eventually receive the message, but we do not assume a known, finite upper bound on message transmission times. Assuming such a bound would be risky in hostile settings, due to the potential of message delays introduced by denial-of-service attacks [23]. While we do assume reliability of communications, only the liveness (but not the safety) of our protocol depends on it.

Each process p_i possesses a private key K_i known only to itself, with which it can digitally sign messages (e.g., [24]). We denote a message $\langle \cdots \rangle$ signed with K_i by $\langle \cdots \rangle_{K_i}$. We assume that each process can obtain the public keys of other processes as needed, with which it can verify the origin of signed messages. As we will see, our protocol does not require all messages to be signed by their senders, but some messages must be signed to ensure that they are not undetectably altered during forwarding.

Each process has a mechanism by which it may come to *suspect* that a process is faulty or correct. These suspicions can be mistaken and can differ between processes. This mechanism is largely independent of our membership protocol, but offers suspicions on which the membership protocol may act to add or remove a process from the group. In practice, this mechanism might be implemented with the help of periodic "heartbeat" messages [9] or hints from a higher-level application. The safety of our protocol does not rely on this mechanism, but liveness does; we discuss this in Section 4.3. If process p suspects q of being faulty, then `faulty`(q) is true at p; otherwise, `correct`(q) holds at p. It is convenient to assume that once a correct p suspects q faulty, it does so forever.

3 Protocol Semantics

As described in Section 1, the goal of a membership protocol, generally speaking, is to enable correct processes to agree on a group of processes that they believe to be currently operational. Beyond this, however, the precise semantics from one membership protocol to the next can vary substantially. Therefore, in this section we more carefully state the semantics of our protocol.

Conceptually, our protocol operates by updating an array V_i at each correct process p_i. The elements of V_i are denoted V_i^x, $x \geq 0$, and V_i^x is called p_i's xth *view* of the group. Initially each V_i^x, $x > 0$, is undefined. The membership protocol updates this array by installing a set of process identifiers as the value of V_i^x for some $x > 0$; once V_i^x is so defined, it is never changed. Views are installed in order of increasing x (i.e., if V_i^x is installed, then only views V_i^y, $y > x$, will subsequently be installed). At any time, if x is the maximum index at which V_i is defined, then V_i^x is p_i's *current view* (or just p_i's *view*) and p_i is said to be *in* view x. In practice, each p_i must retain only its current view, not all elements of V_i. The protocol assumes an initial state in which for some nonempty, finite set P and all correct p_i, if $p_i \in P$ then $V_i^0 = P$, and if $p_i \notin P$ then V_i^0 is undefined. This initial state can be achieved manually by a systems administrator or automatically under an administrator's supervision (see, e.g., [25]).

Our protocol satisfies the following four properties on how views are defined. First, our protocol ensures that for any x, the xth view at each correct process is the same.

Uniqueness. If p_i and p_j are correct and V_i^x and V_j^x are defined, then $V_i^x = V_j^x$.

Uniqueness is common to many membership protocols, including [3], [6], but is also stronger than the ordering semantics of some others. For instance, with the protocol of [7] and the "weak" and "hybrid" protocols of [9], concurrent failures may result in the failed processes being removed from processes' views in different orders.

The second property is also shared with other membership protocols. Intuitively, this property says that views "make sense:" each correct process is a member of its own view and the correct members of its view are eventually aware of their membership in the group.

Validity. If p_i is correct and V_i^x is defined, then $p_i \in V_i^x$ and for all correct $p_j \in V_i^x$, V_j^x is eventually defined.

Note that by Uniqueness, V_j^x, once defined, equals V_i^x. Validity and Uniqueness imply that those correct p_i at which V_i^x is defined are exactly the correct members of all such V_i^x. So, the correct processes with defined xth views intuitively form a group, i.e., a set of processes that mutually believe one another to be members. For convenience, we thus define the xth *group view* V^x to be V_i^x for any correct p_i such that V_i^x is defined. If there is no such p_i, then V^x is undefined. Our protocol defines group views in increasing order: if V^{x+1} is defined, then V^x was previously defined.

While Uniqueness and Validity correspond to properties of several other membership protocols, other membership protocols satisfy them only when processes fail benignly. Our protocol, however, satisfies them even when processes behave maliciously. Moreover, the fact that processes can behave maliciously forces us to add additional features, to prevent faulty processes from manipulating the group membership.

Integrity. If $p \in V^x - V^{x+1}$, then `faulty`(p) held at some correct $q \in V^x$, and if $p \in V^{x+1} - V^x$, then `correct`(p) held at some correct $q \in V^x$.

This property prevents faulty processes from single-handedly *causing* membership changes to occur. Finally, we would like a property to ensure that faulty processes cannot *prevent* membership changes from occurring.

Liveness. If there is a correct $p \in V^x$ such that $\lceil (2|V^x| + 1)/3 \rceil$ correct members of V^x do not suspect p faulty, and a $q \in V^x$ such that `faulty`(q) holds at $\lfloor (|V^x| - 1)/3 \rfloor + 1$ correct members of V^x, then eventually V^{x+1} is defined.

A similar property can be stated for the case "$q \notin V^x$ and `correct`(q) holds at $\lfloor (|V^x| - 1)/3 \rfloor + 1$ correct members of V^x." Intuitively, Liveness says that if enough correct members want to remove a process q and there is some correct member p that is not suspected faulty by enough correct members, then the membership is eventually changed. This property may seem weaker than desired, for two reasons. First, progress relies on accurate failure suspicions about some correct process p. Chandra, et al., have shown that this limitation is fundamental [26]; a protocol satisfying our safety properties cannot be unconditionally live in an asynchronous system. Second, Liveness does not imply that q is eventually removed. In fact, with minor modifications, our protocol does ensure that if for all $y \geq x$, $\lceil (2|V^y| + 1)/3 \rceil$ correct members of V^y do not suspect p faulty and $\lfloor (|V^y| - 1)/3 \rfloor + 1$ correct members of V^y suspect q faulty, then q is eventually removed. For simplicity, however, here we content ourselves with the Liveness guarantee presented above.

4 The Protocol

Our protocol was most directly influenced by the work of Ricciardi and Birman [6], [25], which solves a similar membership problem in asynchronous systems where only crash failures occur. In our protocol we adopt a manager-based protocol structure that is similar to that of [6], [25]. However, our consideration of malicious corruptions of group members, in addition to member crashes, results in a substantially more complex protocol.

Our protocol executes on a per-view basis: when a process in view x installs view $x + 1$, it terminates the protocol for view x and begins the protocol for view $x + 1$. The protocol for each p_i in view x operates under the premise that Uniqueness and Validity are satisfied for processes' xth views, and thus that V^x is well-defined. If this is the case, the protocol ensures that they are satisfied for processes' $(x + 1)$th views. As stated informally in Section 1, however, our protocol requires that at most $\lfloor (|V^x| - 1)/3 \rfloor$ members of V^x are faulty (and thus that at least $\lceil (2|V^x| + 1)/3 \rceil$ members are correct). That is, if one-third of the members of a group view fail, then we cannot ensure that Uniqueness, Validity, Integrity and Liveness will continue to be satisfied, or indeed that the next group view is well-defined. Recall that in Section 3, we assumed views V_i^0 at all p_i that satisfy Uniqueness and Validity; we further require that at least $\lceil (2|V^0| + 1)/3 \rceil$ members of the initial group view V^0 are correct.

Each p_i in view x assigns to each $p \in V_i^x$ a unique *rank* for view x in the set $\{1, \ldots, |V_i^x|\}$, thereby totally ordering the members of V_i^x by rank. Our protocol requires that correct processes in the same view rank processes in the same way. This can be done, e.g., by ranking processes based on a well-known total order of process identifiers or on seniority in the group. In each view V^x, there is a distinguished member called the *manager* that is, by definition, the member with the highest rank (i.e., with rank $|V_i^x|$) at each correct $p_i \in V^x$.

The protocol for each process p_i in view x is presented formally in Appendix A. In the remainder of this section, our goal is to present this protocol in a high-level and intuitive manner, highlighting the basic techniques used and some of the issues that must be addressed. To enable the reader to correlate our discussion to the presentation in Appendix A, however, we annotate our discussion with references to the line numbers of Figs. 5, 6, and 7 in Appendix A.

At its highest level, our protocol executes as follows. In each view V^x, the manager is responsible for suggesting an *update* to the view, which is the name of a process that, based on the recommendations of group members, should be added to or removed from the group. V^{x+1} is obtained by the members of V^x either adopting the manager's suggestion and updating the group membership accordingly, or removing the manager from the group. Our protocol ensures that each correct member of V^x takes the same action; intuitively this is how we achieve Uniqueness.

4.1 Correct Manager

In this section we outline the execution of the protocol in the case in which the manager is correct and is not suspected faulty by correct members. The case in which the

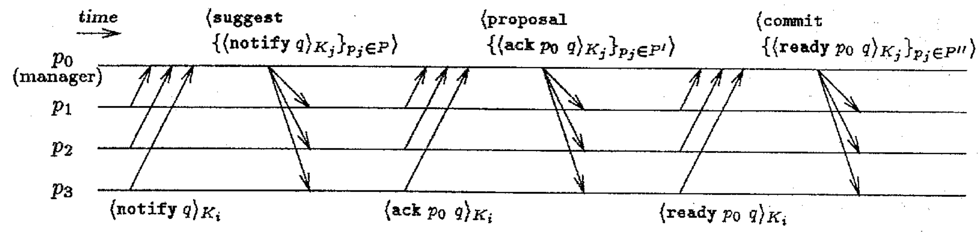

time
$\langle$suggest $\{\langle$notify $q\rangle_{K_j}\}_{p_j \in P}\rangle$
$\langle$proposal $\{\langle$ack $p_0\ q\rangle_{K_j}\}_{p_j \in P'}\rangle$
$\langle$commit $\{\langle$ready $p_0\ q\rangle_{K_j}\}_{p_j \in P''}\rangle$
p_0 (manager)
p_1
p_2
p_3
$\langle$notify $q\rangle_{K_i}$
$\langle$ack $p_0\ q\rangle_{K_i}$
$\langle$ready $p_0\ q\rangle_{K_i}$

Fig. 1. Protocol when manager is correct.

manager is suspected faulty by correct members is discussed in Section 4.2. The protocol in the case we consider here, i.e., the manager is correct and not suspected faulty, is shown in Fig. 1.

As mentioned previously, in each view V^x it is the manager's responsibility to suggest an update, based on the recommendations of group members, to apply to V^x to obtain V^{x+1}. To facilitate this, when a process p_j in view x comes to suspect that some $q \in V_j^x$ is faulty or that some $q \notin V_j^x$ is correct, it sends a notification $\langle$**notify** $q\rangle_{K_j}$ to the manager of V^x (Fig. 5, line 5.5), indicating that it believes that q's membership status should be changed (i.e., that q should be removed from or added to the group). The manager, say p_i, collects notifications from members of V_i^x until for some process q, it has received notifications from $\lfloor(|V_i^x|-1)/3\rfloor+1$ members to change q's status. The number $\lfloor(|V_i^x|-1)/3\rfloor+1$ is significant because, under the assumption that (V^x is well-defined and) at most $\lfloor(|V^x|-1)/3\rfloor$ members of V^x are faulty, it ensures that some correct member of V^x wants to change the status of q.

Having received messages $\{\langle$**notify** $q\rangle_{K_j}\}_{p_j \in P}$ for some $P \subseteq V_i^x$ where $|P| = \lfloor(|V_i^x|-1)/3\rfloor+1$, the manager p_i sends a suggestion $\langle$**suggest** $\{\langle$**notify** $q\rangle_{K_j}\}_{p_j \in P}\rangle$ to the members of V_i^x (Fig. 6, line 6.4). When each process p_j receives this message from the manager, it tests whether the message was created correctly, i.e., if it contains $\{\langle$**notify** $q\rangle_{K_k}\}_{p_k \in P}$ for some q and $P \subseteq V_j^x$ where $|P| = \lfloor(|V_j^x|-1)/3\rfloor+1$. If so, p_j returns to p_i a signed acknowledgment $\langle$**ack** $p_i\ q\rangle_{K_j}$ for p_i's suggestion (line 5.13). In addition, p_j adjusts its state so that it will never send another acknowledgment to p_i in this view (lines 5.11–12).

The manager p_i waits for $\lceil(2|V_i^x|+1)/3\rceil$ acknowledgments for its suggestion (line 6.12). The number $\lceil(2|V_i^x|+1)/3\rceil$ is significant because, if at most $\lfloor(|V^x|-1)/3\rfloor$ members of V^x are faulty, it ensures that a majority of the correct members of V^x have acknowledged the manager. Since a correct process acknowledges only one suggestion from the manager, there can be at most one update for which there are $\lceil(2|V^x|+1)/3\rceil$ acknowledgments. And, if at most $\lfloor(|V^x|-1)/3\rfloor$ members of V^x are faulty, the manager will receive $\lceil(2|V^x|+1)/3\rceil$ acknowledgments.

Upon receiving messages $\{\langle$**ack** $p_i\ q\rangle_{K_j}\}_{p_j \in P}$ where $P \subseteq V_i^x$ and $|P| = \lceil(2|V_i^x|+1)/3\rceil$, the manager p_i sends a proposal $\langle$**proposal** $\{\langle$**ack** $p_i\ q\rangle_{K_j}\}_{p_j \in P}\rangle$ containing these acknowledgments to the members of V_i^x (line 6.13). When a process p_j receives the proposal, it verifies that the proposal was created correctly (line 5.24) and if so, returns $\langle$**ready** $p_i\ q\rangle_{K_j}$ (line 5.27), indicating its readiness to commit the update. Note that even if p_i were faulty, it could not convince a correct process p_j to send $\langle$**ready** $p_i\ q'\rangle_{K_j}$ for some $q' \neq q$, due to p_j's requirement that there be $\lceil(2|V_j^x|+1)/3\rceil$ acknowledgments for q'. Once p_i collects a set of messages $\{\langle$**ready** $p_i\ q\rangle_{K_j}\}_{p_j \in P}$ for some $P \subseteq V_i^x$ where $|P| = \lceil(2|V_i^x|+1)/3\rceil$ (line 6.15), it broadcasts[1] a **commit** message $\langle$**commit** $\{\langle$**ready** $p_i\ q\rangle_{K_j}\}_{p_j \in P}\rangle$ (line 6.16). A process $p_j \in V^x$ that receives this message verifies that it was created correctly (line 5.28) and, if so, installs V_j^{x+1} by adding or removing q (lines 5.31–32). $\lceil(2|V^x|+1)/3\rceil$ **ready** messages are required so that a committed update will be detected if the manager later fails, as is discussed in Section 4.2.

4.2 Faulty Manager

The protocol can become much more complex if the manager is suspected faulty by some correct processes. In this case, some process, called a deputy, may need to take over for the manager and attempt to complete the transition to the next view. The next view may be obtained by removing the manager from the group or, if the manager could have already committed an update to some correct process, by ensuring that all correct members commit that update. In either case, it must be ensured that all correct members commit the same update, even if the deputy is faulty.

A process p_i, which is not the manager, becomes a deputy if enough members suspect all other members with rank higher than p_i of being faulty. To be precise, if a process p_j in view x suspects all members of V_j^x with rank higher than p_i of being faulty, it sends a message $\langle$**deputy** $p_i\rangle_{K_j}$ to p_i, to

1. A *broadcast* message eventually reaches all correct members or, and only if the initiator is faulty, none of them; see Appendix A.

indicate that it thinks p_i should become a deputy (line 5.7). If p_i receives messages $\{\langle \textbf{deputy}\ p_i\rangle_{K_j}\}_{p_j \in P}$ where $P \subseteq V_i^x$ and $|P| = \lfloor (|V_i^x| - 1)/3 \rfloor + 1$ then it initiates the deputy protocol by broadcasting $\langle \textbf{query}\ \{\langle \textbf{deputy}\ p_i\rangle_{K_j}\}_{p_j \in P}\rangle$ to the members of V_i^x (line 6.7). This message shows that some correct member believes that p_i should become a deputy.

In response to this **query** message (if it is properly constructed), each member p_j returns $\langle \textbf{last}\ p_i\ S\rangle_{K_j}$ where S is the set of acknowledgments contained in the last valid **proposal** message it received, or $\emptyset$ if it has not yet received a proposal (line 5.10). p_j also adjusts its state so that it will not respond to the manager or deputies of higher rank than p_i (line 5.9). The set S is returned to convey any update that could have been committed by the manager or a deputy of higher rank than p_i: since $\lceil (2|V^x| + 1)/3 \rceil$ processes must send **ready** messages for an update to be committed (see Section 4.1), if an update was committed, then the acknowledgments for the update were already received at a majority of the correct members of V^x. So, if the deputy p_i receives $\lceil (2|V^x| + 1)/3 \rceil$ **last** messages, at least one of these messages contains a set of acknowledgments for the committed update.

Upon receiving $\lceil (2\ |V_i^x|\ + 1)/3 \rceil$ **last** messages $\{\langle \textbf{last}\ p_i\ S_j\rangle_{K_j}\}_{p_j \in P}$, p_i sends to V_i^x a suggestion $\langle \textbf{suggest}\ \{\langle \textbf{last}\ p_i\ S_j\rangle_{K_j}\}_{p_j \in P}\rangle$ that contains these messages (line 6.10). From this point the protocol continues much like that of Section 4.1, as if p_i had sent a **suggest** message as the manager, but with one major difference. It is simple for a process that receives a manager's **suggest** message to determine the update it should acknowledge—it is just the update in the included **notify** messages (see Section 4.1). In this case, however, a receiving process p must derive, from the messages $\{\langle \textbf{last}\ p_i\ S_j\rangle_{K_j}\}_{p_j \in P}$, an update to acknowledge. This is simple if all **last** messages indicate that all $p_j \in P$ received no prior proposal (in which case p acknowledges the update naming the manager) or the same prior proposal (in which case p acknowledges the update in that proposal). However, these **last** messages may indicate that different processes received *different* last proposals.

As shown in Fig. 2, this could happen even if no processes behave maliciously. In Fig. 2, the manager's proposal is received only by p_1. The first deputy p_6 attempts to install the next view, but fails after sending its **proposal** message to remove the manager. (p_6's messages are also delayed to p_1.) Then, the second deputy p_2 collects **last** messages from the remainder of the group and sends its **suggest** message. Note that the **last** messages in p_2's suggestion contain a set of acknowledgments for q, the update initially proposed by the manager p_0, and a set of acknowledgments for the update p_0. Moreover, it is not difficult to extend this example to one in which some correct process may have actually committed one of these updates and installed its next view. If this occurred and processes $p_1, \ldots, p_5$ acknowledge the wrong update, then Uniqueness could be violated.

Intuitively, given a suggestion

$$\langle \textbf{suggest}\ \{\langle \textbf{last}\ p\ S_j\rangle_{K_j}\}_{p_j \in P}\rangle,$$

a process should acknowledge the update r with the property that for some $p_j \in P$, $S_j = \{\langle \textbf{ack}\ q\ r\rangle_{K_k}\}_{p_k \in Q}$ (for some appropriate Q) and q is the lowest ranked process in the set of all processes q' ranked greater than p such that for some r' and $p_j \in P$, $S_j = \{\langle \textbf{ack}\ q'\ r'\rangle_{K_k}\}_{p_k \in Q_j}$ (for some appropriate Q_j). For instance, in Fig. 2, after receiving p_2's suggestion each process p_j should reply with $\langle \textbf{ack}\ p_2\ p_0\rangle_{K_j}$ since the process that proposed to remove p_0, namely p_6, is the lowest ranked of all processes (with rank greater than p_2) that made a proposal. As we prove in Appendix B, the strategy of acknowledging the update proposed by this lowest ranked proposer ensures that this update will be the same as any update that could have been previously committed to another member. This update is identified in lines 5.15–21 of Fig. 5.

Once a process p_j determines the update q to acknowledge, the protocol continues as in Section 4.1. That is, p_j sends $\langle \textbf{ack}\ p_i\ q\rangle_{K_j}$ to the deputy p_i (line 5.23). Upon receipt of $\lceil (2|V_i^x| + 1)/3 \rceil$ acknowledgments for q, p_i then sends its

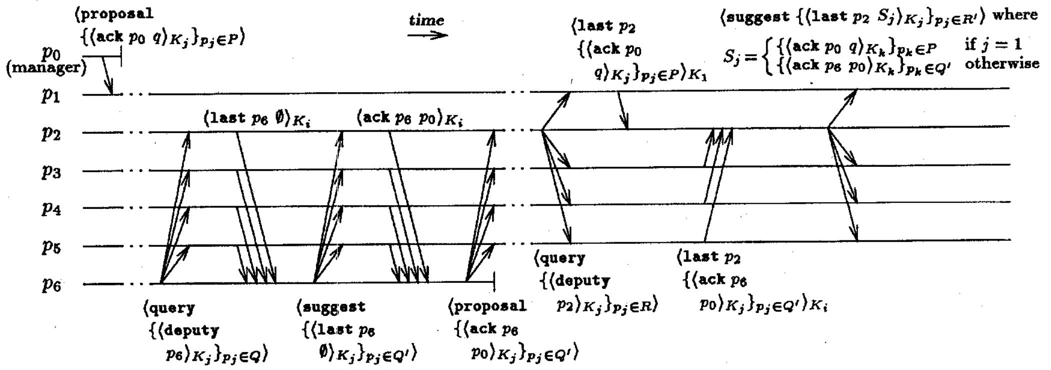

⟨proposal {⟨ack p0 q⟩Kj}pj∈P⟩
time
p0 (manager)
p1
p2
p3
p4
p5
p6
⟨last p6 ∅⟩Ki
⟨ack p6 p0⟩Ki
⟨query {⟨deputy p6⟩Kj}pj∈Q⟩
⟨suggest {⟨last p6 ∅⟩Kj}pj∈Q'⟩
⟨proposal {⟨ack p6 p0⟩Kj}pj∈Q'⟩
⟨last p2 {⟨ack p0 q⟩Kj}pj∈P⟩K1
⟨suggest {⟨last p2 Sj⟩Kj}pj∈R'⟩ where
Sj = {⟨ack p0 q⟩Kk}pk∈P if j = 1; {⟨ack p6 p0⟩Kk}pk∈Q' otherwise
⟨query {⟨deputy p2⟩Kj}pj∈R⟩
⟨last p2 {⟨ack p6 p0⟩Kj}pj∈Q'⟩Ki

Fig. 2. Processes $p_1, \ldots, p_5$ are informed of two different proposals for the next view.

proposal message (line 6.13). When a process p_j receives this proposal, it verifies that it was created correctly (line 5.24) and, if so, sends $\langle$**ready** $p_i\ q\rangle_{K_j}$ to p_i. Once p_i obtains $\lceil(2|V_i^x|+1)/3\rceil$ such **ready** messages, it broadcasts its **commit** message, thereby causing correct processes to add or remove q.

It is worth recalling how our protocol masks malicious behavior by faulty processes. First, a manager's **suggest** message or a deputy's **query** message must contain signed **notify** or **deputy** messages, respectively, from $\lfloor(|V^x|-1)/3\rfloor+1$ members of V^x to be considered valid. This ensures that a process cannot be added or removed without the agreement of at least one correct member. Second, a **proposal** message must contain $\lceil(2|V^x|+1)/3\rceil$ acknowledgments for an update—and thus acknowledgments from a majority of correct processes in V^x—for the proposal to be considered valid. So, it is impossible for a faulty process to send valid proposals for different updates to different processes. Third, if an update is committed to a member, then any valid **suggest** message sent by a subsequent deputy, even if the deputy is faulty, will contain evidence that this update was committed. This is true because $\lceil(2|V^x|+1)/3\rceil$ members must send **ready** messages for the update to be committed, and because the deputy's **suggest** message must contain **last** messages from $\lceil(2|V^x|+1)/3\rceil$ members—and thus a **last** message from some correct member that sent a **ready** message for the update. This ensures that once an update has been committed somewhere, correct processes will acknowledge only the same update, and thus that Uniqueness will be maintained. These arguments are formalized in Appendix B.

4.3 Failure Suspicions

So far, our protocol description has treated failure suspicions as inputs generated independently of the membership protocol, and indeed none of Uniqueness, Validity, or Integrity require any further assumptions regarding failure suspicions. To prove Liveness, however, we must make certain assumptions regarding these suspicions. To see why, suppose that a member p fails and that all correct processes suspect it of being faulty, but the (faulty) manager refuses to suggest that p be removed from the group. Unless correct members come to suspect the manager faulty and remove it, a next view may never be installed. We thus impose the following assumptions on the failure suspicion mechanism.

Eventual Suspicion. Suppose that for some correct p_i in view x and some faulty $r \in V_i^x$, `faulty`(q) holds at p_i for 1) some $q \in V_i^x$, and 2) all $q \in V_i^x$ ranked higher than r. If p_i does not install V_i^{x+1} (for sufficiently long), then eventually `faulty`(r) holds at p_i.

Gossip. If a correct p_i in view x receives

$$\langle \textbf{query}\ \{\langle \textbf{deputy}\ r\rangle_{K_j}\}_{p_j\in P}\rangle$$

for some $P \subseteq V_i^x$, $|P| = \lfloor(|V_i^x|-1)/3\rfloor+1$, then `faulty`$(q)$ becomes true at p_i for all $q \in V_i^x$ ranked higher than r.

Eventual Suspicion prevents correct processes from waiting on faulty managers and deputies forever without suspecting them faulty. Gossip requires a correct process to sometimes adopt failure suspicions about a manager or deputies from another correct process. The necessity of Gossip to Liveness will be made clear in Appendix B.

4.4 Joiner Protocol

In this section, we describe the protocol for a process joining the group. In stating this protocol, it is convenient to assume that a correct process joins the group at most once, or in other words, that a removed but correct process rejoins the group as a new process with a new process identifier. In practice, this can be implemented by composing process identifiers from two components: a static identifier for the process (e.g., its public key) and a value that changes with each of that process' joins (e.g., an incarnation number or nonce identifier).

The main difficulty presented to a joining process is determining the view in which it is first a member. That is, in the protocol described in Sections 4.1 and 4.2, a process p_i installs V_i^{x+1} after receiving a message of the form $\langle$**commit** $\{\langle$**ready** $p\ q\rangle_{K_j}\}_{p_j\in P}\rangle$ for some $P \subseteq V_i^x$ where $|P| = \lceil(2|V_i^x|+1)/3\rceil$. For p_i to interpret or verify the validity of this **commit** message, it must know the contents of V^x, because otherwise it is not able to, e.g., determine if P is of the proper size or form. However, a joining process $p_i \in V^{x+1} - V^x$ may not know the contents of V^x (because V_i^x is not defined). Thus, the joining protocol must take other measures to ensure that p_i will install a proper V_i^{x+1}.

The basis of our solution to this problem is that it suffices for p_i to obtain the contents of *some* past group view V^y where $y \le x$, and the **commit** messages sent in views y through x that tell it how to transform V^y into V^{x+1}. To be able to provide these **commit** messages to joining processes, correct members maintain a *history* set containing, for each prior view, a valid **commit** message sent in that view (line 5.29). Before a correct $p_j \in V^x$ installs a view V_j^{x+1} containing a new member p_i, it sends $\langle$**history** $p_i\ S\rangle$ to p_i where S is its history set, including the **commit** message sent in view x (line 5.30).

The joiner's part of this protocol is shown formally in Fig. 7 of Appendix A. Informally, the protocol begins with the joining process obtaining the contents of some past group view V^y; below we discuss approaches by which this information can be obtained.[2] The joiner then waits to receive a **history** message and, upon receiving one (Fig. 7,

2. Depending on how members are ranked, it may be helpful for the contents of V^y obtained by the joiner to convey the ranks of the members of V^y. For instance, if members are ranked by seniority in the group (in the manner of [6], [25]) and the obtained contents of V^y are ordered by rank, then the joiner can determine the rank of each process in later views, because any two processes in V^y are ranked in the same relative order in later views (until one of them is removed).

line 7.5), extracts **commit** messages from the history and constructs subsequent views V^z, $z > y$ (lines 7.7–16). The joining process, say p_i, continues accepting **history** messages and producing subsequent views to find the first view V^{x+1} that contains its own identifier. It then installs V_i^{x+1} and initiates the normal protocol for that view.

This scheme relies on the ability of a joining process to obtain the contents of a prior group view. There are several possible ways to enable this:

1) A trusted authority (e.g., the group creator or administrator) could deposit with the group members the contents of some group view signed by the authority's private key. Then, the members themselves could send the signed view to a process prior to adding the process to the group. Provided that the process could obtain the authority's public key, the process could verify the validity of the group view and then save the view to distribute to other processes. A variation of this approach is to store the signed view in a replicated database that holds signed views for many groups. Processes could obtain the view from the database if at least one database server is correct. Since many public-key distribution systems employ such databases to distribute public keys (e.g., [27]), this alternative may require little extra mechanism or administrative overhead in a system already employing public key technology.
2) Using the techniques of [16], [19] a secure, fault-tolerant service could be constructed to maintain and distribute recent group views for possibly many groups. Each group's initial view could be stored at the service as part of the group creation, and then copies of **commit** messages for that group could be forwarded to the service to update the group view held in the service. Processes wishing to join a group would first query the service to obtain a group view for the group. This approach, however, may require additional assumptions bounding the number of the service's component servers that could be faulty [16], [19]. Moreover, the techniques of [16] would require processes wishing to join a group to be able to identify and authenticate each of these servers.
3) If there is a known set of nodes on which all members of V^x for all $x \geq 0$ will execute, then the set V^0 can be written to local stable storage on each node as part of the group creation, so that a process joining the group will at least know V^0. Moreover, if each process updates local stable storage when it installs a view, then a joining process may know a more recent view. This approach would work well, say, for a group of server processes executing on a small, static set of server nodes. However, it requires checks to ensure that a faulty process cannot mislead a future joining process on the same node simply by updating local stable storage with an incorrect view.

Depending on how joining processes obtain group views, steps may need to be taken to ensure that the view obtained by a joining process is indeed a *past* group view. For example, in the second approach above, if a process is added to the group and additional views are committed before the process can obtain a view from the service, the process might obtain a view after that in which it was added. Such confusions can be avoided if, e.g., each correct p_j in view x delays sending $\langle \textbf{notify}\ q\rangle_{K_j}$ for some $q \notin V^x$ until q has obtained a view. (For simplicity, such synchronizations are omitted from Figs. 5, 6, and 7.)

As our protocol is presented in Figs. 5, 6, and 7, each member retains a **commit** message for every view update committed (lines 5.29, 7.6). In practice, a **commit** message sent in view x can be discarded when it is known that every correct process that joins in the future will know the contents of V^y for some $y > x$. For instance, in the first approach described above, if the trusted authority periodically updates the signed group view to a more recent view V^y, then the **commit** messages sent in views V^x, $x < y$, can be discarded after each update. Here there is a tradeoff between the amount of state that members must maintain and the frequency with which the authority updates the signed view. However, as experience with group-oriented systems suggests that membership changes infrequently in most applications (e.g., see [20]), storage costs at members should typically be modest even if the signed view is updated infrequently. In the second approach, a process could discard a **commit** message for view x after it is received at the service (and at any $q \in V^{x+1} - V^x$), provided that correct processes p_j in view x refrain from sending $\langle \textbf{notify}\ q\rangle_{K_j}$ for some $q \notin V^x$ until q has obtained V^x. In the third approach, once some V_i^y where $y > x$ has been written to local stable storage at all nodes that can host member processes, all **commit** messages for view x could be discarded from processes' histories.

5 Performance

As just mentioned, experience with current group-oriented systems has shown that membership changes are infrequent for most applications. Based on this, we do not expect our protocol to be the primary factor limiting performance in most applications that use it. Nevertheless, if our protocol is to be useful in a wide range of applications, efficiency will be important, and this weighed heavily in the design of our protocol. For instance, we chose a manager-based protocol structure, versus a symmetric protocol involving more messages (but possibly fewer phases of communication), to minimize message traffic. Moreover, the traffic generated by our protocol as presented in Section 4 and Appendix A can be further reduced by various optimizations. We have omitted these optimizations here, however, for purposes of clarity.

We implemented a prototype of our protocol as part of the Rampart tool kit [22]. Our implementation employs CryptoLib [28] for its cryptographic operations and runs over the Multicast Transport Service [29], which supports point-to-point authenticated channels [14]. Figs. 3 and 4 illustrate the protocol cost in milliseconds for removing a group member with this implementation. The tests described in these figures were performed between user processes running over SunOS 4.1.3 on moderately loaded SPARCstation 10s spanning several networks. In these tests, we used RSA [24] as our digital signature scheme, with 512-bit moduli and public exponents equal to three. Figs. 3 and 4 show average times between the initiation and termination of the protocol at group members in tests in which a nonmanager process was removed via the protocol of Section 4.1 (Fig. 3) and in which the manager was removed via the protocol of Section 4.2 (Fig. 4). Those curves marked "total" show the average elapsed real time between initiation and termination at each member, and the curves marked "CPU" show the average CPU time consumed by the protocol between initiation and termination at each member.

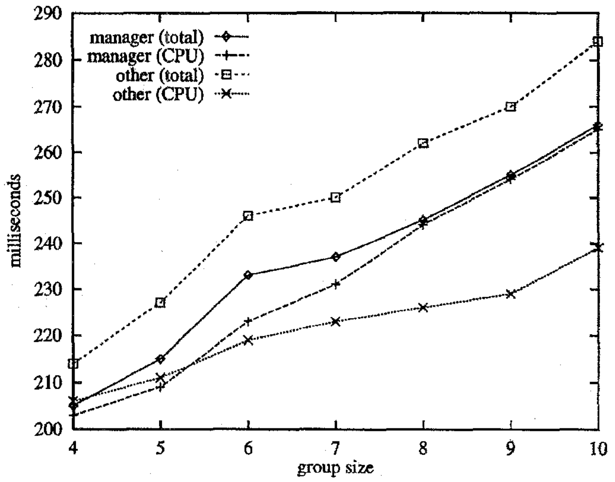


Fig. 3. Protocol cost, correct manager.

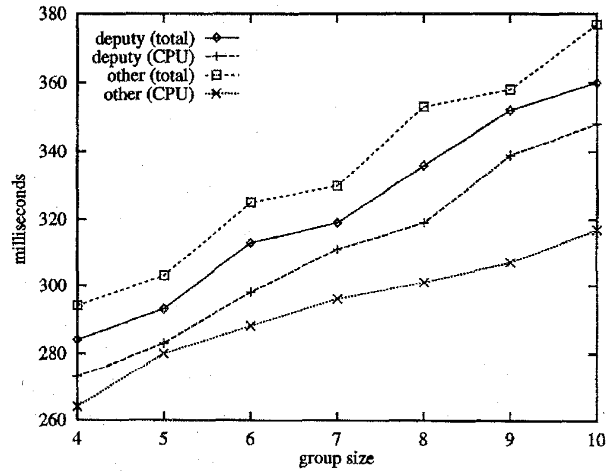


Fig. 4. Protocol cost, faulty manager.

Even though CryptoLib provides a very fast software implementation of RSA (roughly 52 msec for signature generation and 2 msec for signature verification, for the described platform and parameters), RSA operations still accounted for over 70% of managers' and deputies' CPU costs and over 80% of others' CPU costs. Clearly our protocol's performance would benefit from special-purpose processors for performing RSA computations.

In these tests, each member initiated the protocol immediately upon suspecting the eventually-removed process faulty, which, for the purposes of these tests, was triggered by a multicast to the group. So, each member initiated the protocol at approximately the same instance. In reality, the moments at which group members come to suspect a process faulty and to initiate the protocol can vary widely with both the types of failures exhibited and the failure detection mechanisms employed. Therefore, the numbers in these figures should be viewed as protocol costs only, rather than the actual duration between a process failure and its removal from the group.

6 Conclusion

In this paper we presented a group membership protocol for asynchronous distributed systems that tolerates the corruption of group members by a malicious intruder. Our protocol provides strong membership semantics, including a total ordering of membership changes among all correct group members, provided that less than one-third of each group view is faulty. Moreover, these faulty members are powerless to single-handedly alter the group membership or prevent membership changes from occurring.

We have implemented our protocol as part of Rampart, a tool kit for constructing secure and fault-tolerant distributed services. Rampart provides protocols and other support for constructing replicated services that can retain their correctness and availability despite the malicious corruption of some of their component servers. The membership protocol presented in this paper, and the implementation of atomic broadcast that it facilitates, complete a set of techniques that make such a tool kit practical.

Appendix A – Formal Protocol Description

The protocol discussed informally in Sections 4.1 and 4.2 is presented formally in Figs. 5 and 6. For clarity, we have divided the protocol description into those steps that each p_i in view x executes in the role of a "regular member" (Fig. 5) and those that it executes as a manager or deputy (Fig. 6). Nevertheless, Figs. 5 and 6 describe a single protocol, to be executed in its entirety by each p_i in view x. In Fig. 5, *rank*(p) denotes the rank of p for view x, and *mgr* denotes the manager for view x. We will also use this notation in the rest of our discussion. Here we do not describe a particular implementation of a failure-suspicion mechanism, though implementing one satisfying Eventual Suspicion and Gossip is trivial in practice.

In Figs. 5 and 6, the protocol is presented in terms of **if** statements (e.g., lines 5.30–32) and **repeat** statements (e.g., lines 5.3–33). The execution of "**if** C : A_1 **else** : A_2" proceeds in the natural way: if condition C holds then the (possibly compound) statement A_1 is executed, and otherwise A_2 is executed. If C is an existential condition that is true when evaluated, then A_1 is executed with variables instantiated by a witness (i.e., a satisfying assignment of bound variables) for C. The **else** clause can also be omitted, as in the innermost **if** statement of line 5.32. The semantics for

```
repeat
  ▯ $C_1$ : $A_1$
  ▯ $C_2$ : $A_2$
  ⋮
  ▯ $C_n$: $A_n$
```

are that the following step is repeated: some condition C_k is evaluated and, if true, the statement A_k is executed (with variables instantiated by a witness for C_k). The evaluation of C_k and the execution of A_k are atomic, so that no other conditions are evaluated or statements executed concurrently. Evaluating conditions and executing statements as appropriate are repeated until the **repeat** statement is terminated with a **terminate repeat** statement. A **terminate repeat** terminates only the closest encompassing **repeat** statement. We assume that the condition evaluated in each iteration is chosen fairly, so that if a condition is continuously satisfied, then eventually the corresponding statement will be executed (if the **repeat** statement is not terminated). Moreover, if there are several witnesses that continuously satisfy an existential condition, then eventually the corresponding statement will be instantiated and executed with each.

In Fig. 5, the outer **repeat** statement is terminated immediately after the protocol changes to the next view by executing

$$V_i^{x+1} \leftarrow \texttt{installview}(\ldots)$$

in line 5.31 or 5.32. The operation $V_i^{x+1} \leftarrow \texttt{installview}(S)$ installs the view $V_i^{x+1} = S$ and initiates the protocol for view $x + 1$ at the top of Figs. 5 and 6.

Each message sent by a correct process in view x is labeled with x. (We have omitted these labels from the figures to simplify the presentation.) In particular, a digitally signed message contains the label as part of its signed contents. Received messages that are labeled with a view number greater than that of the receiver's current view are buffered until the process installs that view. Received messages that are labeled with a view number less than that of the receiver's current view are immediately discarded and ignored, as are messages that are labeled for one view but that contain messages labeled for a different view.

Messages can be sent and received by one of two interfaces. A process can send a message m to a process using a **send** (e.g., line 5.5); a message can be sent to each member of a view by specifying the view as the destination (e.g., line 6.4). The predicate $\texttt{rcvd}(p, m)$ in Figs. 5 and 6 is true if p_i received the message m on the communication channel from p. The second interface enables messages to be sent by *broadcast* to a view (e.g., line 6.16). We assume that this broadcast is implemented by a protocol that ensures that 1) if a correct process broadcasts m to V^x, then all correct members of V^x receive the broadcast m, and 2) if any correct member of V^x receives a broadcast m to V^x, then all correct members of V^x receive the broadcast m. Property 2) distinguishes broadcasting from a process simply sending a message to each member of a view, because in the latter, if the sender is faulty then the message might, e.g., reach only some members. Broadcasting can be implemented in our system model using a simple message diffusion protocol; see [30]. In Figs. 5 and 6, the predicate $\texttt{brcvd}(m)$ is true if p_i received a broadcast message m.

Fig. 7 is the formal description of the protocol for a joining process p_i. As in Section 4.4, our description assumes that a correct p_i joins the group only once. In Fig. 7, $\texttt{oldview}$ and $\texttt{oldviewno}$ denote, respectively, the contents of a past view V^y and its view number y that were obtained by p_i. Ways to obtain these were discussed in Section 4.4. The $\texttt{label}(m)$ operation in line 7.8, where $m = \langle \textbf{commit}\, \{\langle \textbf{ready}\ q\ r\rangle_{K_j}\}_{p_j \in P}\rangle$, returns the number of the view in which m and the messages $\{\langle \textbf{ready}\ q\ r\rangle_{K_j}\}_{p_j \in P}$ were sent according to their view labels (which are required to be the same). Other than this, view labels are ignored in Fig. 7. Note that the outer **repeat** statement in Fig. 7 terminates only after both the first view V_i^{x+1} containing p_i has been installed (i.e., $\texttt{done}$ = true) and histories from at least $\lfloor (|V^x| - 1) / 3 \rfloor + 1$ members of V^x have been incorporated into $\texttt{history}$ (line 7.17). This ensures that $\texttt{history}$ will be sufficiently complete to enable other joining processes to construct their own joining views from it. The history should be complete in this regard prior to sending it in line 5.30; we have omitted this synchronization from the figures for simplicity.

APPENDIX B – CORRECTNESS

In this appendix we prove that our protocol satisfies Uniqueness, Validity, Integrity and Liveness. Our proofs employ the assumptions that at most $\lfloor (|V^x| - 1) / 3 \rfloor$ members of each view V^x are faulty and that a correct joining process obtains a valid $\texttt{oldview}$ and $\texttt{oldviewno}$ (lines 7.1–2). In what follows, ranks are for view x, and all messages are assumed to be labeled for view x (see Appendix A).

Of the four properties, the proof for Uniqueness is the most complex, as it must address the issues raised in Section 4.2 of ensuring that if an update is committed to some members by the manager or a deputy, then no different update can be committed to other members by a future deputy. The full proof is by induction on views; the following lemma is the key to the induction step.

LEMMA 1. *If a process receives* $\langle \textbf{commit}\, \{\langle \textbf{ready}\ p\ r\rangle_{K_j}\}_{p_j \in P}\rangle$ *where* $P \subseteq V^x$ *and* $|P| = \lceil (2|V^x| + 1) / 3 \rceil$, *then the only* r' *for which a correct* $p_i \in V^x$ *will send* $\langle \textbf{ack}\ q\ r'\rangle_{K_i}$, *where* $\texttt{rank}(q) < \texttt{rank}(p)$, *is* $r' = r$.

PROOF. Suppose a process receives

$$\langle \textbf{commit}\, \{\langle \textbf{ready}\ p\ r\rangle_{K_j}\}_{p_j \in P}\rangle$$

where $P \subseteq V^x$ and $|P| = \lceil (2|V^x| + 1) / 3 \rceil$. Then, each process p_k in some majority of the correct processes in V^x sent $\langle \textbf{ready}\ p\ r\rangle_{K_k}$ in line 5.27 and assigned $\texttt{lastproposal} = \{\langle \textbf{ack}\ p\ r\rangle_{K_j}\}_{p_j \in P_k}$ in line 5.26, for some $P_k \subseteq V^x$ where $|P_k| = \lceil (2|V^x| + 1) / 3 \rceil$.

```
(5.1)   $protocolstate \leftarrow 3|V_i^x|$
(5.2)   $lastproposal \leftarrow \emptyset$
(5.3)   repeat
(5.4)   [] $\exists p((p \in V_i^x \wedge faulty(p)) \vee (p \notin V_i^x \wedge correct(p)))$
(5.5)     : send $\langle \texttt{notify}\ p\rangle_{K_i}$ to $mgr$
(5.6)   [] $\exists p \in V_i^x(p \neq mgr \wedge \forall q \in V_i^x(rank(q) > rank(p) \Rightarrow faulty(q)))$
(5.7)     : send $\langle \texttt{deputy}\ p\rangle_{K_i}$ to $p$
(5.8)   [] $\exists p \in V_i^x, P \subseteq V_i^x(brcvd(\langle \texttt{query}\ \{\langle \texttt{deputy}\ p\rangle_{K_j}\}_{p_j \in P}\rangle) \wedge 3rank(p) < protocolstate \wedge |P| = \lfloor(|V_i^x|-1)/3\rfloor + 1)$
(5.9)     : $protocolstate \leftarrow 3rank(p)$
(5.10)      send $\langle \texttt{last}\ p\ lastproposal\rangle_{K_i}$ to $p$
(5.11)  [] $\exists p, P \subseteq V_i^x(rcvd(mgr, \langle \texttt{suggest}\ \{\langle \texttt{notify}\ p\rangle_{K_j}\}_{p_j \in P}\rangle) \wedge$
                $3rank(mgr) - 1 < protocolstate \wedge |P| = \lfloor(|V_i^x|-1)/3\rfloor + 1)$
(5.12)    : $protocolstate \leftarrow 3rank(mgr) - 1$
(5.13)      send $\langle \texttt{ack}\ mgr\ p\rangle_{K_i}$ to $mgr$
(5.14)  [] $\exists p \in V_i^x, P \subseteq V_i^x, \{S_j\}_{p_j \in P}(rcvd(p, \langle \texttt{suggest}\ \{\langle \texttt{last}\ p\ S_j\rangle_{K_j}\}_{p_j \in P}\rangle) \wedge$
                             $3rank(p) - 1 < protocolstate \wedge |P| = \lceil(2|V_i^x|+1)/3\rceil)$
(5.15)    : $lowestrank \leftarrow |V_i^x| + 1$
(5.16)      $lowestupdate \leftarrow mgr$
(5.17)      repeat
(5.18)      [] true :  if $\exists p_j \in P, q \in V_i^x, r, Q \subseteq V_i^x(S_j = \{\langle \texttt{ack}\ q\ r\rangle_{K_k}\}_{p_k \in Q} \wedge$
                                   $rank(p) < rank(q) < lowestrank \wedge |Q| = \lceil(2|V_i^x|+1)/3\rceil)$
(5.19)                   : $lowestrank \leftarrow rank(q)$
(5.20)                     $lowestupdate \leftarrow r$
(5.21)              else :  terminate repeat
(5.22)      $protocolstate \leftarrow 3rank(p) - 1$
(5.23)      send $\langle \texttt{ack}\ p\ lowestupdate\rangle_{K_i}$ to $p$
(5.24)  [] $\exists p \in V_i^x, q, P \subseteq V_i^x(rcvd(p, \langle \texttt{proposal}\ \{\langle \texttt{ack}\ p\ q\rangle_{K_j}\}_{p_j \in P}\rangle) \wedge$
                       $3rank(p) - 2 < protocolstate \wedge |P| = \lceil(2|V_i^x|+1)/3\rceil)$
(5.25)    : $protocolstate \leftarrow 3rank(p) - 2$
(5.26)      $lastproposal \leftarrow \{\langle \texttt{ack}\ p\ q\rangle_{K_j}\}_{p_j \in P}$
(5.27)      send $\langle \texttt{ready}\ p\ q\rangle_{K_i}$ to $p$
(5.28)  [] $\exists p \in V_i^x, q, P \subseteq V_i^x(brcvd(\langle \texttt{commit}\ \{\langle \texttt{ready}\ p\ q\rangle_{K_j}\}_{p_j \in P}\rangle) \wedge |P| = \lceil(2|V_i^x|+1)/3\rceil)$
(5.29)    : $history \leftarrow history \cup \{\langle \texttt{commit}\ \{\langle \texttt{ready}\ p\ q\rangle_{K_j}\}_{p_j \in P}\rangle\}$
(5.30)      if $q \notin V_i^x$ : send $\langle \texttt{history}\ q\ history\rangle$ to $q$
(5.31)                   $V_i^{x+1} \leftarrow installview(V_i^x \cup \{q\})$
(5.32)        else    : if $p_i \neq q$ :  $V_i^{x+1} \leftarrow installview(V_i^x - \{q\})$
(5.33)      terminate repeat
```

Fig. 5. Member protocol for each process p_i in view x.

```
(6.1)   $mdstate \leftarrow \texttt{begin}$
(6.2)   repeat
(6.3)   [] $mdstate = \texttt{begin} \wedge \exists q, P \subseteq V_i^x(\forall p_j \in P(rcvd(p_j, \langle \texttt{notify}\ q\rangle_{K_j})) \wedge |P| = \lfloor(|V_i^x|-1)/3\rfloor + 1)$
(6.4)     : send $\langle \texttt{suggest}\ \{\langle \texttt{notify}\ q\rangle_{K_j}\}_{p_j \in P}\rangle$ to $V_i^x$
(6.5)       $mdstate \leftarrow \texttt{sentsugg}$
(6.6)   [] $mdstate = \texttt{begin} \wedge \exists P \subseteq V_i^x(\forall p_j \in P(rcvd(p_j, \langle \texttt{deputy}\ p_i\rangle_{K_j})) \wedge |P| = \lfloor(|V_i^x|-1)/3\rfloor + 1)$
(6.7)     : broadcast $\langle \texttt{query}\ \{\langle \texttt{deputy}\ p_i\rangle_{K_j}\}_{p_j \in P}\rangle$ to $V_i^x$
(6.8)       $mdstate \leftarrow \texttt{sentquery}$
(6.9)   [] $mdstate = \texttt{sentquery} \wedge \exists P \subseteq V_i^x, \{S_j\}_{p_j \in P}(\forall p_j \in P(rcvd(p_j, \langle \texttt{last}\ p_i\ S_j\rangle_{K_j})) \wedge |P| = \lceil(2|V_i^x|+1)/3\rceil)$
(6.10)    : send $\langle \texttt{suggest}\ \{\langle \texttt{last}\ p_i\ S_j\rangle_{K_j}\}_{p_j \in P}\rangle$ to $V_i^x$
(6.11)      $mdstate \leftarrow \texttt{sentsugg}$
(6.12)  [] $mdstate = \texttt{sentsugg} \wedge \exists q, P \subseteq V_i^x(\forall p_j \in P(rcvd(p_j, \langle \texttt{ack}\ p_i\ q\rangle_{K_j})) \wedge |P| = \lceil(2|V_i^x|+1)/3\rceil)$
(6.13)    : send $\langle \texttt{proposal}\ \{\langle \texttt{ack}\ p_i\ q\rangle_{K_j}\}_{p_j \in P}\rangle$ to $V_i^x$
(6.14)      $mdstate \leftarrow \texttt{sentprop}$
(6.15)  [] $mdstate = \texttt{sentprop} \wedge \exists q, P \subseteq V_i^x(\forall p_j \in P(rcvd(p_j, \langle \texttt{ready}\ p_i\ q\rangle_{K_j})) \wedge |P| = \lceil(2|V_i^x|+1)/3\rceil)$
(6.16)    : broadcast $\langle \texttt{commit}\ \{\langle \texttt{ready}\ p_i\ q\rangle_{K_j}\}_{p_j \in P}\rangle$ to $V_i^x$
(6.17)      terminate repeat
```

Fig. 6. Manager/deputy protocol for each process p_i in view x.

```
(7.1)   $viewno \leftarrow oldviewno$
(7.2)   $view \leftarrow oldview$
(7.3)   $done \leftarrow \text{false}$
(7.4)   repeat
(7.5)   [] $\exists p, S(rcvd(p, \langle \texttt{history}\ p_i\ S\rangle))$
(7.6)     : $history \leftarrow history \cup S$
(7.7)       repeat
(7.8)       [] true :  if $done = \text{false} \wedge \exists q \in view, r, m \in history, P \subseteq view(m = \langle \texttt{commit}\ \{\langle \texttt{ready}\ q\ r\rangle_{K_j}\}_{p_j \in P}\rangle \wedge$
                                          $|P| = \lceil(2|view|+1)/3\rceil \wedge label(m) = viewno)$
(7.9)                     : if $p_i = r$ :  broadcast $m$ to $view$
(7.10)                                  $V_i^{viewno+1} \leftarrow installview(view \cup \{r\})$
(7.11)                                  $done \leftarrow \text{true}$
(7.12)                                  terminate repeat
(7.13)                        else   : if $r \in view$ :  $view \leftarrow view - \{r\}$
(7.14)                                     else       :  $view \leftarrow view \cup \{r\}$
(7.15)                                  $viewno \leftarrow viewno + 1$
(7.16)                 else : terminate repeat
(7.17)      if $done = \text{true} \wedge \exists P \subseteq view, \{S_j\}_{p_j \in P}(|P| = \lfloor(|view|-1)/3\rfloor + 1 \wedge \forall p_j \in P(rcvd(p_j, \langle \texttt{history}\ p_i\ S_j\rangle) \wedge S_j \subseteq history))$
(7.18)           : terminate repeat
```

Fig. 7. Protocol for a joining process p_i.

Now suppose that a correct process $p_i \in V^x$ sends $\langle \mathbf{ack}\ q\ r' \rangle_{K_i}$ where $rank(q) < rank(p)$. To do this, p_i must have received a message

$$\left\langle \mathbf{suggest}\ \{ \langle \mathbf{last}\ q\ S_j \rangle_{K_j} \}_{p_j \in Q} \right\rangle$$

where $Q \subseteq V^x$ and $|Q| = \lceil (2|V^x| + 1)/3 \rceil$. Because $|Q| = \lceil (2|V^x| + 1)/3 \rceil$, each process p_k in some majority of the correct processes in V^x must have sent $\langle \mathbf{last}\ q\ S_k \rangle_{K_k}$. Moreover, at least one of the correct processes p_k in that majority must have previously set $lastproposal = \{ \langle \mathbf{ack}\ p\ r \rangle_{K_j} \}_{p_j \in P_k}$ as described above.

We now show by induction on $rank(p) - rank(q)$ that $r' = r$. So, for the base case, suppose that $rank(p) - rank(q) = 1$. Then for some correct $p_k \in P \cap Q$, $S_k = \{ \langle \mathbf{ack}\ p\ r \rangle_{K_j} \}_{p_j \in P_k}$. Moreover, since p is the lowest ranked process with rank greater than q and there could not be any $S_l = \{ \langle \mathbf{ack}\ p\ r'' \rangle_{K_j} \}_{p_j \in P_l}$ where $P_l \subseteq V^x$, $|P_l| = \lceil (2|V^x| + 1)/3 \rceil$, and $r'' \neq r$, it follows (from 5.17–21) that $r' = r$.

Now suppose that $rank(p) - rank(q) > 1$, and consider the lowest ranked process q' such that $rank(q') > rank(q)$ and there exists a $p_k \in Q$ and a r'' such that $S_k = \{ \langle \mathbf{ack}\ q'\ r'' \rangle_{K_j} \}_{p_j \in Q'}$ where $Q' \subseteq V^x$ and $|Q'| = \lceil (2|V^x| + 1)/3 \rceil$. Since there is some correct process in $P \cap Q$, it is guaranteed that there is some such q' and, moreover, that $rank(p) \geq rank(q')$. If $rank(p) = rank(q')$ (i.e., $p = q'$), then the result follows as in the base case. Otherwise, since $rank(q') > rank(q)$ we know that $rank(p) - rank(q') < rank(p) - rank(q)$ and so $r'' = r$ by the induction hypothesis; the result follows. □

THEOREM 1. *This protocol satisfies Uniqueness.*

PROOF. The proof is by induction on views. The induction step goes as follows. For a correct process p_i to execute $V_i^{x+1} \leftarrow installview(\ldots)$, it must receive a message $\langle \mathbf{commit}\ \{ \langle \mathbf{ready}\ p\ r \rangle_{K_j} \}_{p_j \in P} \rangle$, where $P \subseteq V^x$ and $|P| = \lceil (2|V^x| + 1)/3 \rceil$, that commits the update r to apply to the xth view. Consider the $p \in V^x$ of largest rank such that some process receives $\langle \mathbf{commit}\ \{ \langle \mathbf{ready}\ p\ r \rangle_{K_j} \}_{p_j \in P} \rangle$, for some P where $P \subseteq V^x$ and $|P| = \lceil (2|V^x| + 1)/3 \rceil$. Since a correct $p_i \in V^x$ sends $\langle \mathbf{ready}\ p\ r \rangle_{K_i}$ for at most one update r, it is not possible for a different correct process to receive $\langle \mathbf{commit}\ \{ \langle \mathbf{ready}\ p\ r' \rangle_{K_j} \}_{p_j \in P'} \rangle$, where $P' \subseteq V^x$, $|P'| = \lceil (2|V^x| + 1)/3 \rceil$, and $r' \neq r$. Moreover, Lemma 1 says that the only update value r'' for which a correct process $p_i \in V^x$ will create $\langle \mathbf{ack}\ q\ r'' \rangle_{K_i}$ where $rank(q) < rank(p)$, or thus $\langle \mathbf{ready}\ q\ r'' \rangle_{K_i}$ is $r'' = r$. □

THEOREM 2. *This protocol satisfies Validity.*

PROOF. By the conditions guarding the $installview$ operations on lines 5.31, 5.32, and 7.10, V_i^{x+1} is defined at a correct process p_i only if $p_i \in V_i^{x+1}$. We have to show that V_j^{x+1} is eventually defined at all correct $p_j \in V_i^{x+1}$. This is done by induction on views: Our induction hypothesis is that if V_i^x is defined at any correct p_i, then V_j^x is defined at all correct $p_j \in V_i^x$.

So, suppose that V_i^{x+1} is defined at a correct p_i, and consider any correct $p_j \in V_i^{x+1}$. In order for V_i^{x+1} to be defined, p_i must receive a valid **commit** message sent to view x containing **ready** messages sent by correct processes in view x. Thus, if V_i^{x+1} is defined, then V_k^x must be defined at some correct p_k and, by the induction hypothesis, either V_j^x is defined or $p_j \notin V^x$. If V_j^x is defined, then because the **commit** message sent in view x is broadcast to the members of V^x (lines 6.16, 7.9), V_j^{x+1} will be defined. If $p_j \notin V^x$, then p_j will eventually receive a **history** message from p_i (line 5.30), which will cause p_j to install V_j^{x+1} (line 7.10). □

THEOREM 3. *This protocol satisfies Integrity.*

PROOF. Suppose that $p \in V^x - V^{x+1}$. Then, at least $\lfloor (|V^x| - 1)/3 \rfloor + 1$ members of V^x sent either **notify** messages indicating that p should be removed or, if p was the manager of V^x, **deputy** messages indicating that some member ranked lower than p should become a deputy. Since there are at most $\lfloor (|V^x| - 1)/3 \rfloor$ faulty members of V^x and since correct members send **notify** and **deputy** messages in accordance with their failure suspicions, it follows that some correct member of V^x suspected p faulty. The argument for $p \in V^{x+1} - V^x$ is similar. □

THEOREM 4. *This protocol satisfies Liveness.*

PROOF. Suppose there is a correct $p \in V^x$ such that $\lceil (2|V^x| + 1)/3 \rceil$ correct members of V^x do not suspect p faulty. Then, no member with rank lower than p can generate a valid **query** message containing $\lfloor (|V^x| - 1)/3 \rfloor + 1$ **deputy** messages. Therefore, if p is the manager and sends a **suggest** message, or if p acts as a deputy and sends a **query** message, then each correct member of V^x will reply to p (if it has not already installed a new view) and p will complete the protocol and install a new view.

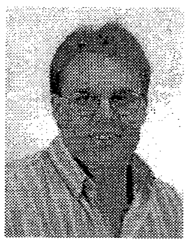
Now suppose there is a process that some set $P \subseteq V^x$ of correct processes, $|P| = \lfloor (|V^x| - 1)/3 \rfloor + 1$, wants to remove. We show by induction on rank that if V^{x+1} is not installed for sufficiently long, then eventually each member of P suspects each $q \in V^x$, $rank(q) > rank(p)$, faulty. Thus, if p is not the manager, then p eventually receives enough **deputy** messages to send a **query**, which, by the argument above, causes V^{x+1} to be installed. For the induction, consider any $q \in V^x$, $rank(q) > rank(p)$, and suppose that each member of P suspects each $r \in V^x$, $rank(r) > rank(q)$, faulty. If q is faulty, then Eventual Suspicion gives us the result. If q is correct but does not succeed in installing V^{x+1}, then some correct member must have received $\langle \mathtt{query}\ \{\langle \mathtt{deputy}\ r\rangle_{K_j}\}_{p_j \in Q}\rangle$ for some r, $rank(q) > rank(r) \geq rank(p)$, and some $Q \subseteq V^x$, $|Q| = \lfloor (|V^x| - 1)/3 \rfloor + 1$. Because **query** messages are broadcast, all correct members receive $\langle \mathtt{query}\ \{\langle \mathtt{deputy}\ r\rangle_{K_j}\}_{p_j \in Q}\rangle$ and, by Gossip, suspect q faulty. □

Acknowledgments

We thank Jack Lacy and Robbert van Renesse for their assistance in using CryptoLib and the Multicast Transport Service, respectively. We also thank Dalia Malki and the anonymous referees for providing helpful suggestions.

References

[1] K.P. Birman and T.A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Computer Systems*, vol. 5, no. 1, pp. 47–76, Feb. 1987.
[2] B.A. Coan and G. Thomas, "Agreeing on a leader in real-time," in *Proc. 11th Real-Time Systems Symp.*, pp. 166–172, Dec. 1990.
[3] F. Cristian, "Reaching agreement on processor group membership in synchronous distributed systems," *Distributed Computing*, vol. 4, pp. 175–187, 1991.
[4] H. Kopetz, G. Grünsteidl, and J. Reisinger, "Fault-tolerant membership service in a synchronous distributed real-time system," *Dependable Computing for Critical Applications*, A. Avižienis and J.C. Laprie, eds., pp. 411–429. Springer-Verlag, 1991.
[5] L.E. Moser, P.M. Melliar-Smith, and V. Agrawala, "Membership algorithms for asynchronous distributed systems," *Proc. 11th Int'l Conf. Distributed Computing Systems*, pp. 480–488, May 1991.
[6] A.M. Ricciardi and K.P. Birman, "Using process groups to implement failure detection in asynchronous environments," *Proc. 10th ACM Symp. Principles of Distributed Computing*, pp. 341–351, Aug. 1991.
[7] S. Mishra, L.L. Peterson, and R.D. Schlicting, "A membership protocol based on partial order," *Dependable Computing for Critical Applications*. J.F. Meyer and R.D. Schlicting, eds., vol. 2, pp. 309–331. Springer-Verlag, 1992.
[8] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella, "Fast message ordering and membership using a logical token-passing ring," *Proc. 13th Int'l Conf. Distributed Computing Systems*, May 1993.
[9] F. Jahanian, A. Fakhouri, and R. Rajkumar, "Processor group membership protocols: Specification, design and implementation," *Proc. 12th Symp. Reliable Distributed Systems*, pp. 2–11, Oct. 1993.
[10] N.P. Kronenberg, H. M. Levy, and W. D. Strecker, "VAX clusters: A closely-coupled distributed system," *ACM Trans. Computer Systems*, vol. 4, no. 2, pp. 130–146, May 1986.
[11] F. Cristian, B. Dancey, and J. Dehn, "Fault-tolerance in the advanced automation system," *Proc. 20th Int'l Symp. Fault-Tolerant Computing*, pp. 6–17, June 1990.
[12] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A communication sub-system for high availability," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing*, pp. 76–84, July 1992.
[13] K. Ilgun, "USTAT: A real-time intrusion detection system for UNIX," *Proc. 1993 IEEE Symp. Research in Security and Privacy*, pp. 16–28, May 1993.
[14] M.K. Reiter, K.P. Birman, and R. van Renesse, "A security architecture for fault-tolerant systems," *ACM Trans. Computer Systems*, vol. 12, no. 4, pp. 340–371, Nov. 1994.
[15] K. Marzullo, "Tolerating failures of continuous-valued sensors," *ACM Trans. Computer Systems*, vol. 8, no. 4, pp. 284–304, Nov. 1990.
[16] F.B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.
[17] M.K. Franklin and M. Yung, "The varieties of secure distributed computation," *Proc. of Sequences II, Methods in Comm., Security and Computer Science*, pp. 392–417, June 1991.
[18] Y. Desmedt, "Threshold cryptography," *European Trans. Telecommunications and Related Technologies*, vol. 5, no. 4, pp. 449–457, July 1994.
[19] M.K. Reiter and K.P. Birman, "How to securely replicate services," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 3, pp. 986–1009, May 1994.
[20] K.P. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Trans. Computer Systems*, vol. 9, no. 3, pp. 272–314, Aug. 1991.
[21] M.K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in Rampart," *Proc. Second ACM Conf. Computer and Comm. Security*, pp. 68–80, Nov. 1994.
[22] M.K. Reiter, "The Rampart toolkit for building high-integrity services," *Theory and Practice in Distributed Systems. Lecture Notes in Computer Science 938*. K.P. Birman, F. Mattern, and A. Schiper, eds., pp. 99–110, Springer-Verlag, 1995.
[23] V.L. Voydock and S.T. Kent, "Security mechanisms in high-level network protocols," *ACM Computing Surveys*, vol. 15, no. 2, pp. 135–171, June 1983.
[24] R.L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Comm. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
[25] A.M. Ricciardi and K.P. Birman, "Process membership in asynchronous environments," Tech. Report 93-1328, Dept. of Computer Science, Cornell Univ., Feb. 1993.
[26] T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," Tech. Report 95-1548, Dept. of Computer Science, Cornell Univ., Oct. 1995.
[27] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Trans. Computer Systems*, vol. 10, no. 4, pp. 265–310, Nov. 1992.
[28] J.B. Lacy, D.P. Mitchell, and W.M. Schell, "CryptoLib: Cryptography in software," *Proc. Fourth USENIX Security Workshop*, pp. 1–17, Oct. 1993.
[29] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson, "Reliable multicast between microkernels," *Proc. USENIX Microkernels and Other Kernel Architectures Workshop*, Apr. 1992.
[30] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," *Distributed Systems*. S. Mullender, ed., 2nd edition, ch. 5, pp. 97–145. Addison-Wesley, 1993.



Michael K. Reiter received the BS degree in mathematical sciences from the University of North Carolina at Chapel Hill in 1989, and the MS and PhD degrees in computer science from Cornell University in 1991 and 1993, respectively. In 1993, he joined AT&T Bell Laboratories as a principal investigator/member of technical staff. His research interests include security and fault-tolerance in distributed systems.