A Secure Group Membership Protocol

Michael K. Reiter

AT&T Bell Laboratories, Holmdel, New Jersey
reiter@research.att.com

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 high-integrity distributed services that we are presently implementing.

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., [3, 5, 6, 13, 19, 23, 18, 2, 12]) and have been implemented in numerous experimental and commercial systems (e.g., [14, 3, 7, 1, 18]). 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 the first 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 singlehandedly effect changes to the group membership or prevent needed changes from occurring. Our protocol achieves these guarantees in a fully 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., [11]). Moreover, the ability of our protocol to tolerate the corruption of group members sets it apart from previous security work in group-oriented systems (e.g., [22, 20]), which focuses on securing group semantics and communication against attacks from outside the group only.

There are many reasons for developing 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., [17, 26]). Use of our protocol to maintain membership information would prevent these attacks.

A second motivation for our protocol is that it facilitates the realisation of other security technologies. For example, the need for group-oriented cryptographic controls and secure group computing techniques has long been understood in the cryptographic and distributed systems communities, and this has given rise to much research in these areas (see [10, 8] for surveys). 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 [21]. This technique requires that client requests be issued to servers by an *atomic broadcast* protocol, which ensures that all correct servers receive requests in the same order. However, it has been shown that due to the inherent difficulty of detecting failures in distributed systems, there is no deterministic protocol that achieves *consensus*, or thus atomic broadcast, even in systems that can suffer only a single crash failure [9]. Assuming that only crash failures occur, several systems have circumvented this impossibility result with the help of a membership protocol (e.g., [4, 1]). With our membership protocol and similar techniques, we can circumvent this impossibility result even when some processes behave maliciously. Our protocol thus completes a set of mechanisms that make the techniques of [21] practical, and we are presently implementing a toolkit, called Rampart, for building high-integrity services using these techniques. Due to space limitations, however, we defer discussion of Rampart to a future paper.

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 sketch our correctness proofs 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 behaves according to its specification is said to be *correct*. A process may *fail* either benignly (by prematurely halting) or maliciously. In the latter case, the process can behave in any fashion whatsoever, limited only by the assumptions stated below, and is said to be *corrupt*.

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 [29]. 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 [29]. 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., [25]). 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 p_i maintains a local set V_i of process identifiers, which is called its *view* of the group; it is this set that our membership protocol will update. Because the set V_i changes over time, it makes sense to talk about the x-th view (i.e., version of V_i) at p_i, which we denote V_i^x, for each $x \geq 0$. For some values of x, V_i^x can be undefined at p_i because an x-th view is never installed. In particular, it will be convenient to allow the first view installed at a process joining the group to be V_i^x for some $x > 0$, so that views V_i^y, $y < x$, would be undefined. If V_i^x is p_i's current view, we say that p_i is *in* view x. We assume an initial state in which there is some nonempty, finite set P such that for 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., [24] and also Section 4.3). We assume that at least $\lceil (2|P| + 1)/3 \rceil$ members of P are correct. As we will discuss in Section 4 and Appendix B, this is necessary for our protocol's correctness.

In addition to its view, each process has a mechanism by which it may come to *suspect* that another process is faulty or correct. These suspicions can be mistaken and can differ between processes. This mechanism is independent of the membership protocol and does not affect processes' views. Rather, it simply offers suspicions on which the membership protocol may act, to add or remove a process from processes' views. In real systems, this mechanism might be implemented with the help of periodic "heartbeat" messages [12] 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 Appendix B. If process p suspects q of being faulty, then *faulty*(q) is true at p. Otherwise, *correct*(q) holds at p.

3 Protocol requirements

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 some properties that our protocol satisfies. Some of these properties are common to a number of membership protocols, and others are a result of the need to guard against a malicious intruder. These semantics facilitate the implementation of a variety of group-based mechanisms and, in particular, an atomic broadcast protocol for use in hostile settings, as discussed in Section 1.

First, our protocol ensures that for any x, the x-th 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 [6, 23], but is also stronger than the ordering semantics of some others. For instance, with the protocol of [18] and the "weak" and "hybrid" protocols of [12], 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 x-th 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 x-th *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.

While Uniqueness and Validity correspond to requirements of several other membership protocols, other membership protocols satisfy them only when processes fail benignly. Our protocol, however, must satisfy them even when processes behave maliciously. Moreover, the fact that processes can behave maliciously forces us to add additional requirements, to prevent corrupt 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 corrupt processes from single-handedly *causing* membership changes to occur. Finally, we would like a liveness requirement to ensure that corrupt processes cannot *prevent* membership changes from occurring. Here we state our liveness guarantee informally, as follows.

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 process $q \in V^x$ (resp., $q \notin V^x$) such that *faulty*(q) (resp., *correct*(q)) holds at $\lfloor (|V^x| - 1)/3 \rfloor + 1$ correct members of V^x, then eventually V^{x+1} is defined.

Intuitively, Liveness says that if sufficiently many correct members want to add or remove a process q and there is some correct member p that is not suspected faulty by sufficiently many correct members, then the membership is eventually changed. This property may seem weaker than desired, as we might prefer to also know that q is eventually added or 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 (if $q \in V^x$) or operational (if $q \notin V^x$), then q is eventually added or removed. For simplicity, however, here we content ourselves with the Liveness guarantee presented above.

4 The protocol

Our protocol was most heavily influenced by that of [23, 24], 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 [23, 24]. 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' x-th 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 2, we assumed views V_i^0 at all p_i that satisfy Uniqueness and Validity, and 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* in the set $\{1,\ldots,|V_i^x|\}$, thereby totally ordering the members 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 upon a well-known total order on process identifiers or upon 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 a 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 Figures 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.[1] 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 manager is suspected faulty by correct members is discussed in Section 4.2. The protocol in the case in which the manager is correct and not suspected faulty is shown in Figure 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 member $p_j \in V^x$ comes to suspect that another member q is faulty or that a non-member q is operational, it sends a notification $\langle\texttt{notify}\ q\rangle_{K_j}$ to the manager (Figure 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 group members 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 $\{\langle\texttt{notify}\ q\rangle_{K_j}\}_{p_j\in P}$ for some $P \subseteq V_i^x$ where $|P| = \lfloor(|V_i^x|-1)/3\rfloor+1$, the manager p_i sends a suggestion $\langle\texttt{suggest}\ \{\langle\texttt{notify}\ q\rangle_{K_j}\}_{p_j\in P}\rangle$ to the members of V_i^x (Figure 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 $\{\langle\texttt{notify}\ q\rangle_{K_h}\}_{p_h\in P}$ 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 acknowledgement $\langle\texttt{ack}\ p_i\ q\rangle_{K_j}$ for p_i's suggestion (line 5.13). In addition, p_j adjusts its state so that it will never send another acknowledgement to p_i in this view (lines 5.11–12).

The manager p_i waits for $\lceil(2|V_i^x|+1)/3\rceil$ acknowledgements 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

[1]Our protocol can easily be modified to accommodate updates consisting of multiple joining and leaving processes, although for simplicity, here we treat only one process at a time.

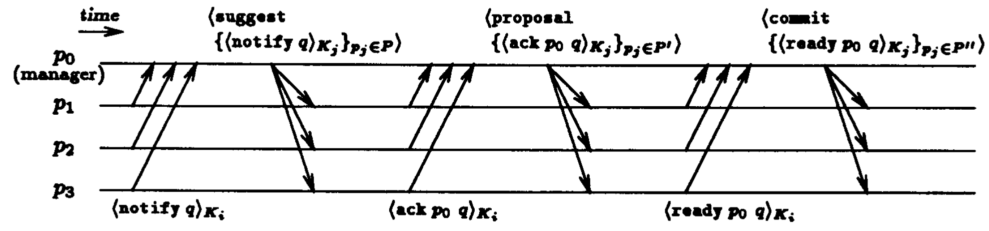

time
$\langle$suggest $\{\langle$notify $q\rangle_{K_j}\}_{p_j \in P}\rangle$
$\langle$proposal $\{\langle$ack p_0 $q\rangle_{K_j}\}_{p_j \in P'}\rangle$
$\langle$commit $\{\langle$ready p_0 $q\rangle_{K_j}\}_{p_j \in P''}\rangle$
p_0 (manager)
p_1
p_2
p_3
$\langle$notify $q\rangle_{K_i}$
$\langle$ack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

Figure 1: Protocol when manager is correct.

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 $\lceil(2|V^x|+1)/3\rceil$ acknowledgements. And, if at most $\lfloor(|V^x|-1)/3\rfloor$ members of V^x are faulty, the manager will receive $\lceil(2|V^x|+1)/3\rceil$ acknowledgements.

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 acknowledgements 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 corrupt, 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 $\lceil(2|V_j^x|+1)/3\rceil$ acknowledgements 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[2] 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 corrupt.

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 member p_j suspects all members with rank higher than p_i of being faulty, it sends a message $\langle$deputy $p_i\rangle_{K_j}$ to p_i, to indicate that it thinks p_i should become a deputy (line 5.7). If p_i receives messages $\{\langle$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 sending $\langle$query $\{\langle$deputy $p_i\rangle_{K_j}\}_{p_j \in P}\rangle$ to the group (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 $\langle$last p_i $S\rangle_{K_j}$ where S is the set of acknowledgements 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 were committed, then the acknowledgements 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 acknowledgements for the committed update.

Upon receiving $\lceil(2|V_i^x|+1)/3\rceil$ last messages $\{\langle$last p_i $S_j\rangle_{K_j}\}_{p_j \in P}$, p_i sends to the group a sug-

[2] As discussed in Appendix A, this broadcast must ensure that the commit message reaches all correct members, if it reaches any of them. This can be implemented very efficiently in our system model (see Appendix A).

gestion $\langle$suggest $\{\langle$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 $\{\langle$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 Figure 2, this could happen even if no processes behave maliciously. In Figure 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 acknowledgements for q, the update initially proposed by the manager p_0, and a set of acknowledgements 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 $\langle$suggest $\{\langle$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 $p_j \in P$, $S_j = \{\langle$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 $p_j \in P$, $S_j = \{\langle$ack q' $r'\rangle_{K_k}\}_{p_k \in Q_j}$ (for some appropriate Q_j). For instance, in Figure 2, after receiving p_2's suggestion each process p_j should reply with $\langle$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 Figure 5.

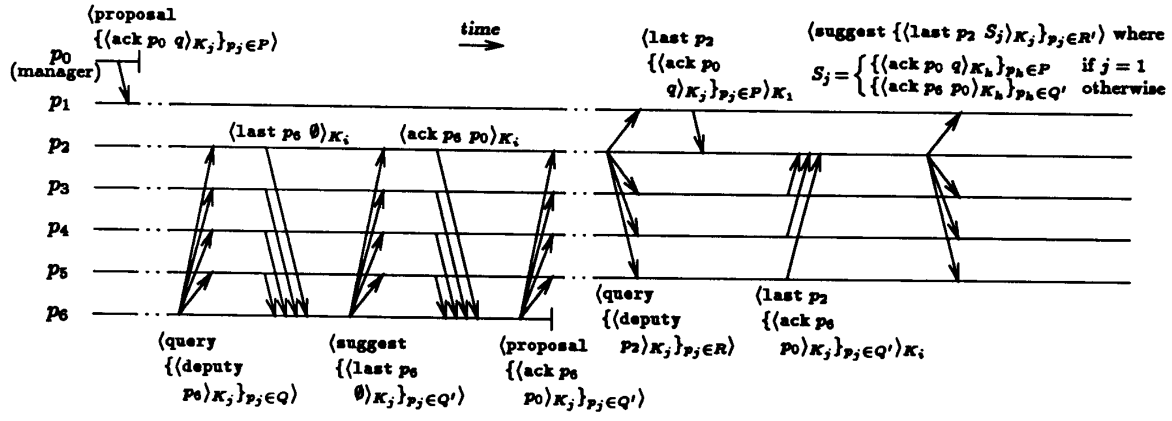
Once a process p_j determines the update q to acknowledge, the protocol continues as in Section 4.1. That is, p_j sends $\langle$ack p_i $q\rangle_{K_j}$ to the deputy p_i (line 5.23). Upon receipt of $\lceil(2|V_i^x| + 1)/3\rceil$ acknowledgements for q, p_i then sends its proposal message (line 6.13). 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.

Our protocol masks malicious behavior by corrupt processes, and it is worth recalling how this is done. 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 $\lceil(2|V^x| + 1)/3\rceil$ acknowledgements for an update—and thus acknowledgements from a majority of correct processes in V^x—for the proposal to be considered valid. Therefore, it is impossible for a corrupt 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 corrupt, 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 Joiner protocol

In the protocol described in Sections 4.1 and 4.2, 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 a 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 may not know the contents of V^x (because V_i^x is not defined), and so we must take other measures to ensure that p_i will install a proper V_i^{x+1}.

In this section we present a simple approach to address this problem. The basis of this approach is that



⟨proposal {⟨ack p_0 q⟩$_{K_j}$}$_{p_j \in P}$⟩
p_0 (manager)
time
p_1
p_2
p_3
p_4
p_5
p_6
⟨last p_6 ∅⟩$_{K_i}$
⟨ack p_6 p_0⟩$_{K_i}$
⟨query {⟨deputy p_6⟩$_{K_j}$}$_{p_j \in Q}$⟩
⟨suggest {⟨last p_6 ∅⟩$_{K_j}$}$_{p_j \in Q'}$⟩
⟨proposal {⟨ack p_6 p_0⟩$_{K_j}$}$_{p_j \in Q'}$⟩
⟨last p_2 {⟨ack p_0 q⟩$_{K_j}$}$_{p_j \in P}$⟩$_{K_1}$
⟨suggest {⟨last p_2 S_j⟩$_{K_j}$}$_{p_j \in R'}$⟩ where
$S_j = \begin{cases} \{\langle \text{ack } p_0\ q\rangle_{K_h}\}_{p_h \in P} & \text{if } j = 1 \\ \{\langle \text{ack } p_6\ p_0\rangle_{K_h}\}_{p_h \in Q'} & \text{otherwise} \end{cases}$
⟨query {⟨deputy p_2⟩$_{K_j}$}$_{p_j \in R}$⟩
⟨last p_2 {⟨ack p_6 p_0⟩$_{K_j}$}$_{p_j \in Q'}$⟩$_{K_i}$

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

it suffices for p_i to obtain the contents of *some* past group view V^y where $y \leq x$, 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 ⟨history S⟩ 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 Figure 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.[3] The joiner then waits to receive a history message and, upon receiving one (Figure 7, line 7.4), extracts commit messages from the history and constructs as many views V^z, $z > y$, as it can (lines 7.6–13).

The joiner does not make use of only one history message, but rather employs as many as it receives. The reason for this is that while a corrupt process cannot undetectably provide a false history, it can provide a history that is "too short", i.e., that does not reflect all the views up to the latest. To see why this can be problematic, suppose that when a process p is added to the group in the x-th view, a corrupt member provides to it a history that enables p to construct only V^y for some $y < x$. If p accepted no more history messages, then it would never come up-to-date with the other processes and would possibly remain stuck in that view until removed. It is not difficult to verify that, in this way, corrupt processes could starve p from ever becoming a participating member of the group. Processing all received history messages prevents this problem.

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

[3] Depending on how members are ranked, it may be helpful for the contents of the view 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 [23, 24]) 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).

(e.g., [27, 16]), this alternative may require little extra mechanism or administrative overhead in a system already employing public key technology.

2. Using the techniques of [26, 21], a high-integrity 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, requires additional assumptions bounding the number of the service's component servers that could be corrupted [26, 21]. Moreover, the techniques of [26] would require processes wishing to join a group to learn the complete membership of the server group prior to using the service.

3. If it is known that for all $x \geq 0$, $V^x \subseteq P$ for some known finite set P, then the set V^0 can be written to local stable storage at each member of P as part of the group creation, so that a process joining the group will at least know V^0. Moreover, if each $p_i \in P$ writes the contents of V_i^y to local stable storage when it installs V_i^y, then a joining process will know a more recent view if it were previously a member. This approach would work well, say, for a group of servers whose members are drawn from a small, static set P.

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 $p_j \in V^x$ delays sending $\langle \texttt{notify}\ q \rangle_{K_j}$ for some $q \notin V^x$ until q has obtained a view. (For simplicity, such synchronizations are omitted from Figures 5–7.)

As our protocol is presented in Figures 5–7, each member retains a `commit` message for every view update committed (lines 5.29, 7.5). 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 [4]), storage costs at members should typically be modest even if the signed view is updated infrequently. In the second approach, a process could discard the `commit` message for view x after this message is received at the service. While this could force a joining process to obtain a group view from the service multiple times (if the membership changes after the process obtains a view but before it can join the group), this is unlikely to be a significant problem in practice. In the third approach, once each $p_i \in P$ has installed (and written to local stable storage) some V_i^y where $y > x$, 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 optimisations (e.g., only $\lfloor (|V^x| - 1)/3 \rfloor + 1$ members of V^x must be designated to send `history` messages to joining processes). We have omitted these optimisations here, however, for purposes of clarity.

We implemented a prototype of our protocol as part of the Rampart effort mentioned in Section 1. Our implementation employs CryptoLib [15] for its cryptographic operations and runs over the Multicast Transport Service [28], which supports point-to-point authenticated channels [20]. Figures 3–4 illustrate the protocol cost in milliseconds (ms) for removing a group member with this implementation. These figures show average times between the initiation and termination of the protocol at group members in the cases in which a non-manager process is removed via the protocol of Section 4.1 (Figure 3) and in which the manager is

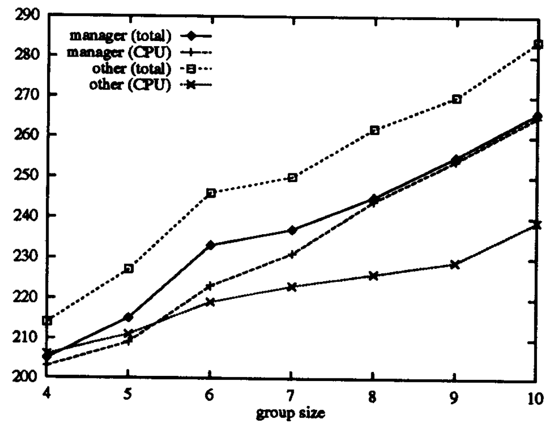


Figure 3: Protocol cost (ms), correct manager

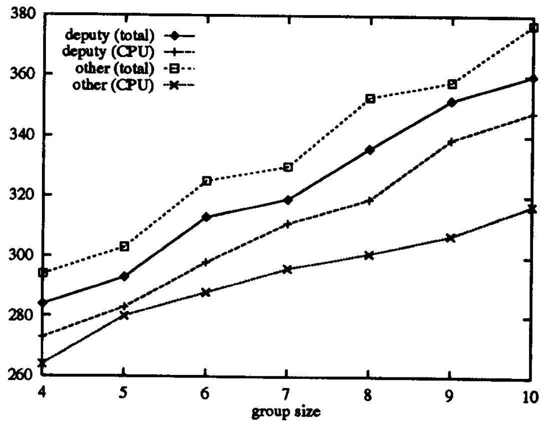


Figure 4: Protocol cost (ms), faulty manager

removed via the protocol of Section 4.2 (Figure 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.

The tests described in Figures 3–4 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 [25] as our public key cryptosystem, with 512-bit moduli and public exponents equal to three. Even though CryptoLib provides a very efficient software implementation of RSA (roughly 52ms for signature generation and 2ms 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 and ongoing work

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 singlehandedly alter the group membership or prevent membership changes from occurring.

The primary focus of our present work is completing the implementation of Rampart, a toolkit for constructing high-integrity distributed services that is based upon the protocol described in this paper. This toolkit will provide protocols and other support for constructing replicated services that can retain their integrity 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 it facilitates, complete a set of techniques that make such a toolkit practical. An initial version of Rampart is nearing completion, and we will report on this effort in a subsequent paper.

Acknowledgements

We thank Jack Lacy and Robbert van Renesse for their assistance in using CryptoLib and the Multicast Transport Service, respectively. We also thank Ran Atkinson, Matt Blaze, Jack Lacy, and the anonymous referees for presentational suggestions.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A commmunication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[2] 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. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

[3] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb. 1987.

[4] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug. 1991.

[5] B. A. Coan and G. Thomas. Agreeing on a leader in real-time. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 166–172, Dec. 1990.

[6] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[7] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in the advanced automation system. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 6–17, June 1990.

[8] Y. Desmedt. Threshold cryptosystems. In *Proceedings of AUSCRYPT '92*, 1992.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[10] M. K. Franklin and M. Yung. Varieties of secure distributed computing. In *Proceedings of Sequences II, Methods in Communications, Security and Computer Science*, pages 392–417, June 1991.

[11] K. Ilgun. USTAT: A real-time intrusion detection system for UNIX. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, pages 16–28, May 1993.

[12] F. Jahanian, A. Fakhouri, and R. Rajkumar. Processor group membership protocols: Specification, design and implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 2–11, Oct. 1993.

[13] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avižienis and J. C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 411–429. Springer-Verlag, 1991.

[14] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. VAXclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[15] J. B. Lacy, D. P. Mitchell, and W. M. Schell. CryptoLib: Cryptography in software. In *Proceedings of the 4th USENIX Security Workshop*, pages 1–17, Oct. 1993.

[16] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[17] K. Marzullo. Tolerating failures of continuous-valued sensors. *ACM Transactions on Computer Systems*, 8(4):284–304, Nov. 1990.

[18] S. Mishra, L. L. Peterson, and R. D. Schlicting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlicting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, 1992.

[19] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 480–488, May 1991.

[20] M. K. Reiter. *A Security Architecture for Fault-Tolerant Systems*. PhD thesis, Cornell University, Aug. 1993.

[21] M. K. Reiter and K. P. Birman. How to securely replicate services. To appear in *ACM Transactions on Programming Languages and Systems*, 1994.

[22] M. K. Reiter, K. P. Birman, and L. Gong. Integrating security in a group oriented distributed system. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, pages 18–32, May 1992.

[23] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–351, Aug. 1991.

[24] A. M. Ricciardi and K. P. Birman. Process membership in asynchronous environments. Technical Report 93-1328, Department of Computer Science, Cornell University, Feb. 1993.

[25] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.

[26] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[27] J. J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 232–244, May 1991.

[28] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX Microkernels and Other Kernel Architectures Workshop*, Apr. 1992.

[29] V. L. Voydock and S. T. Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15(2):135–171, June 1983.

A Formal protocol description

The protocol discussed informally in Sections 4.1 and 4.2 is presented formally in Figures 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" (Figure 5) and those that it executes as a manager or deputy (Figure 6). Nevertheless, Figures 5 and 6 describe a single protocol, to be executed in its entirety by each p_i in view x. In

```
(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)   [] $\exists p \in V_i^x, P \subseteq V_i^x(rcvd(p, \langle \texttt{query}\ \{\langle \texttt{deputy}\ p\rangle_{K_j}\}_{p_j \in P}\rangle) \wedge 3rank(p) < protocolstate \wedge |P| = \lfloor(|V_i^x| - 1)/3\rfloor + 1)$
(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)  [] $\exists q \in V_i^x, p, r, P \subseteq V_i^x(rcvd(p, \langle \texttt{commit}\ \{\langle \texttt{ready}\ q\ r\rangle_{K_j}\}_{p_j \in P}\rangle) \wedge |P| = \lceil(2|V_i^x| + 1)/3\rceil)$
(5.29)     : $history \leftarrow history \cup \{\langle \texttt{commit}\ \{\langle \texttt{ready}\ q\ r\rangle_{K_j}\}_{p_j \in P}\rangle\}$
(5.30)       if $r \notin V_i^x$ : send $\langle \texttt{history}\ history\rangle$ to $r$
(5.31)                  $V_i^{x+1} \leftarrow installview(V_i^x \cup \{r\})$
(5.32)         else    : if $p_i \neq r$ : $V_i^{x+1} \leftarrow installview(V_i^x - \{r\})$
(5.33)       terminate repeat
```

Figure 5: Member protocol for each process p_i in view x.

Figure 5, *rank*(p) denotes the rank of p, and *mgr* denotes the manager. We will also use this notation in the remainder of our discussion.

In Figures 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. 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. 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 satisfy an existential condition, then eventually the corresponding statement will be instantiated and executed with each.

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

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

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

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

in line 5.31 or 5.32. The operation $V_i^{x+1} \leftarrow installview(S)$ installs the view $V_i^{x+1} = S$ and initiates the protocol for view $x+1$ at the top of Figures 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. The predicate $rcvd(p, m)$ in Figures 5 and 6 is true iff process p_i received the message m from p, and m and all messages it contains are labeled for view x.

In contrast to other messages, `commit` messages are ***broadcast*** to the members of V^x (e.g., line 6.16). We assume that this broadcast is implemented by a distributed protocol that ensures that if any correct member of V^x receives a valid `commit` message, then all correct members do, even if the process initiating the broadcast is faulty. For this reason, broadcasting is different than a process p_i simply sending a message m to each member of V_i^x, which we abbreviate by "**send** m **to** V_i^x" (e.g., line 6.4), because in the latter, if p_i is faulty then m might, e.g., reach only some correct members. In our present implementation, a broadcast of a `commit` message is implemented by the initiator sending the message to the entire group, and when each $p \in V^x$ first receives a valid `commit` message for view x, p forwards this message to the $\lfloor (|V^x| - 1)/3 \rfloor + 1$ members q satisfying $rank(p) < rank(q) \leq rank(p) + \lfloor (|V^x| - 1)/3 \rfloor + 1$ or $0 < rank(q) \leq \lfloor (|V^x| - 1)/3 \rfloor + 1 - (|V^x| - rank(p))$. More efficient implementations may be possible using negative acknowledgement schemes.

In Figure 7 is the formal description of a joining process' protocol, which was discussed in Section 4.3. In that figure, *oldview* and *oldviewnumber* denote, respectively, the contents of a past view V^y and its view number y that were obtained by the joining process. Approaches to obtaining these were discussed in Section 4.3. The $label(m)$ operation in line 7.7, where $m = \langle \texttt{commit}\ \{\langle \texttt{ready}\ q\ r \rangle_{K_j}\}_{p_j \in P} \rangle$, returns the number of the view in which m and the messages $\{\langle \texttt{ready}\ q\ r \rangle_{K_j}\}_{p_j \in P}$ were sent according to their view labels (which are required to be the same). Other than this, view labels are ignored in Figure 7.

For reasons discussed in Section 4.3, the outer **repeat** statement of Figure 7 does not terminate. Since this protocol may install a view (line 7.11), though, thereby initiating the protocol of Figures 5 and 6 for that view, this protocol may execute in parallel with the protocol of Figures 5 and 6. Thus, these executions must be coordinated so that, e.g., views are not installed multiple times. The necessary synchronisations are obvious and have been omitted for simplicity.

```
(7.1)   $x \leftarrow oldviewnumber$
(7.2)   $view \leftarrow oldview$
(7.3)   repeat
(7.4)   | $\exists p, S(rcvd(p, \langle \texttt{history}\ S \rangle))$
(7.5)     : $history \leftarrow history \cup S$
(7.6)       repeat
(7.7)       | true :  if $\exists q \in view, r, m \in history, P \subseteq view(m = \langle \texttt{commit}\ \{\langle \texttt{ready}\ q\ r \rangle_{K_j}\}_{p_j \in P} \rangle \wedge$
                                        $|P| = \lceil (2|view| + 1)/3 \rceil \wedge label(m) = x)$
(7.8)                   : broadcast $m$ to $view$
(7.9)                     if $r \in view$ : $view \leftarrow view - \{r\}$
(7.10)                       else       : $view \leftarrow view \cup \{r\}$
(7.11)                    if $p_i \in view$ : $V_i^{x+1} \leftarrow installview(view)$
(7.12)                    $x \leftarrow x + 1$
(7.13)               else : terminate repeat
```

Figure 7: Protocol for a joining process p_i.

B Correctness

In this appendix we sketch the proofs that our protocol satisfies Uniqueness, Validity, Integrity and Liveness. 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 and employs the assumption that at most $\lfloor (|V^x| - 1)/3 \rfloor$ members of each view V^x are faulty. In what follows, all messages are assumed to be labeled for view x (see Appendix A). The following lemma is the key to the inductive step.

Lemma 1 *If a process receives* $\langle \texttt{commit}\ \{\langle \texttt{ready}\ p\ r \rangle_{K_j}\}_{p_j \in P} \rangle$ *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* $\langle \texttt{ack}\ q\ r' \rangle_{K_i}$, *where* $rank(q) < rank(p)$, *is* $r' = r$.

Proof. Suppose a process receives $\langle \texttt{commit}\ \{\langle \texttt{ready}\ p\ r \rangle_{K_j}\}_{p_j \in P} \rangle$ 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 $\langle \texttt{ready}\ p\ r \rangle_{K_k}$ in line 5.27 and assigned $lastproposal = \{\langle \texttt{ack}\ p\ r \rangle_{K_j}\}_{p_j \in P_k}$ in line 5.26, for some $P_k \subseteq V^x$ where $|P_k| = \lceil (2|V^x| + 1)/3 \rceil$.

Now suppose that a correct process $p_i \in V^x$ sends $\langle \texttt{ack}\ q\ r' \rangle_{K_i}$ where $rank(q) < rank(p)$. To do this, p_i must have received a message $\langle \texttt{suggest}\ \{\langle \texttt{last}\ q\ S_j \rangle_{K_j}\}_{p_j \in Q} \rangle$ 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 $\langle \texttt{last}\ q\ S_k \rangle_{K_k}$. Moreover, at least one of the correct processes p_k in that majority must have previously set $lastproposal = \{\langle \texttt{ack}\ p\ r \rangle_{K_j}\}_{p_j \in P_k}$ 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 $p_k \in P \cap Q$, $S_k = \{\langle \texttt{ack}\ p\ r \rangle_{K_j}\}_{p_j \in P_k}$. Moreover, since p is the lowest ranked process with rank greater than q and there could not be any $S_l = \{\langle \texttt{ack}\ p\ r'' \rangle_{K_j}\}_{p_j \in P_l}$ 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 $S_k = \{\langle \texttt{ack}\ q'\ r'' \rangle_{K_j}\}_{p_j \in Q'}$ 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. (Sketch.) The full proof is by induction on views. The core of the induction step is as follows. For a correct process p_i to execute $V_i^{x+1} \leftarrow installview(\ldots)$, it must receive a message $\langle \texttt{commit}\ \{\langle \texttt{ready}\ p\ r \rangle_{K_j}\}_{p_j \in P} \rangle$, where $P \subseteq V^x$ and $|P| = \lceil (2|V^x| + 1)/3 \rceil$, that commits the update r to apply to the x-th view. Consider the $p \in V^x$ of largest rank such that some process receives $\langle \texttt{commit}\ \{\langle \texttt{ready}\ p\ r \rangle_{K_j}\}_{p_j \in P} \rangle$, 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 $\langle \texttt{ready}\ p\ r \rangle_{K_i}$ for at most one update r, it is not possible for a different correct process to receive $\langle \texttt{commit}\ \{\langle \texttt{ready}\ p\ r' \rangle_{K_j}\}_{p_j \in P'} \rangle$, 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 $\langle \texttt{ack}\ q\ r'' \rangle_{K_i}$ where $rank(q) < rank(p)$, or thus $\langle \texttt{ready}\ q\ r'' \rangle_{K_i}$, is $r'' = r$. □

Theorem 2 *This protocol satisfies Validity.*

Proof. (Sketch.) By the conditions guarding the *installview* operations on lines 5.31, 5.32, and 7.11, 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 for any correct $p_j \in V_i^{x+1}$, V_j^{x+1} is eventually defined. This is done by induction on views: our induction hypothesis is that if V_i^x is defined and $p_j \in V_i^x$, then V_j^x is defined.

First suppose that V_i^{x+1} is installed in line 5.31 or 5.32 of Figure 5. Because the broadcast by which the `commit` message is disseminated ensures that all correct members of V_i^x receive it, if $p_j \in V_i^x$, then p_j installs V_j^{x+1}. If $p_j \notin V_i^x$, then because communication is reliable, p_j will eventually receive a `history` message from p_i sent in line 5.30, which will cause it to install V_j^{x+1} (line 7.11). Now suppose that V_i^{x+1} is installed in line 7.11 of Figure 7. Since p_i broadcasts to V^x a `commit` message sent in view x (line 7.8), each correct $p_j \in V^x$ will eventually receive a `commit` message for view x and install V_j^{x+1}, and will send a `history` message to any joining p_j so that it will install a proper V_j^{x+1}. □

Theorem 3 *This protocol satisfies Integrity.*

Proof. (Sketch.) 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 were 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. □

The remaining theorem to prove is that this protocol satisfies Liveness. In order to do this, however, we must constrain the criteria by which processes do or do not suspect other processes of being faulty. To see why, suppose that a member p fails and that all correct processes suspect it of being faulty, but the (corrupt) manager refuses to suggest that p be removed from the group. Unless correct members come to suspect the manager of being faulty and remove the manager, a next view may never be installed. For this and other reasons, we informally stipulate that $faulty(p)$ hold at a correct process r if r desires a change in the group membership, $faulty(q)$ holds at r for all members q such that $rank(q) > rank(p)$, and sufficiently long passes without a change in the group membership occurring. Given this stipulation, Liveness follows easily.

Theorem 4 *This protocol satisfies Liveness.*

Proof. (Sketch.) 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, and a process q such that $\lfloor (|V^x| - 1)/3 \rfloor + 1$ correct members of V^x want to add or remove q. Since $\lceil (2|V^x| + 1)/3 \rceil$ correct members of V^x do not suspect p faulty, 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 be able to complete the protocol and install a new view. Liveness then follows from the above stipulation on failure suspicions, because the $\lfloor (|V^x| - 1)/3 \rfloor + 1$ correct members of V^x desiring to add or remove q will either provide p with enough `notify` messages to make a suggestion as manager or, if a member with rank higher than p does not succeed in committing a new view, with enough `deputy` messages to make a query as deputy. □