Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart

Michael K. Reiter

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

Abstract

Reliable and atomic group multicast have been proposed as fundamental communication paradigms to support secure distributed computing in systems in which processes may behave maliciously. These protocols enable messages to be multicast to a group of processes, while ensuring that all honest group members deliver the same messages and, in the case of atomic multicast, deliver these messages in the same order. We present new reliable and atomic group multicast protocols for asynchronous distributed systems. We also describe their implementation as part of Rampart, a toolkit for building *high-integrity* distributed services, i.e., services that remain correct and available despite the corruption of some component servers by an attacker. To our knowledge, Rampart is the first system to demonstrate reliable and atomic group multicast in asynchronous systems subject to process corruptions.

1 Introduction

In practice, the only support for secure interprocess communication in most distributed systems, if any, is secure channels [32]. Secure channels by themselves, however, provide little support for secure distributed computing, especially when global security policies must be met despite the malicious behavior of some system components. Both practical and theoretical research on secure distributed services (e.g., [22]), verifiable secret sharing (e.g., [20]), secure distributed elections (e.g., [12]), and more general forms of secure distributed computing (see [9] for a survey) indicate that meeting global security requirements can involve substantial cryptographic and/or communication mechanisms over and above secure channels. If the techniques developed in this research are to be realized in practice, support for them must be explored.

Reliable and atomic group multicast are two communication paradigms proposed to support some forms of secure distributed computing (e.g., [11, 6, 20, 28, 2, 22]; also see [9]). Each of these is a protocol by which a process can multicast a message to a group of processes. Reliable multicast, also known as Byzantine agreement [15], ensures that all honest group members (i.e., members that obey the protocol) deliver the same messages, even in the face of malicious multicast initiators. Atomic multicast adds the property that honest members deliver these messages in the same order. While reliable and atomic multicast protocols tolerant of only benign failures have been the focus of much systems research (e.g., [5, 16, 17, 13, 10, 3, 1]), relatively little has been done to extend these results to the more stringent requirements of security.

In this paper we present new reliable and atomic group multicast protocols tolerant of malicious processes. We have implemented these protocols as part of Rampart, a toolkit for building *high-integrity* distributed services, i.e., services that remain correct and available despite the corruption of some component servers by an attacker. Rampart was motivated by an effort in which we are examining ways to support a variety of security technologies in loosely coupled, large-scale distributed systems. This effort exposed the need for high-integrity services to support security-critical tasks, such as cryptographic key distribution, management and enforcement of global access control policy, and secure audit. We developed Rampart to facilitate the construction of these services to be both highly available and highly secure, even to the extent of tolerating the penetration of some servers by attackers.

Rampart supports the construction of high-integrity services with the techniques of [28], including the security extensions of [22]. Briefly, in this approach a service is implemented with multiple identical, deterministic servers, initialized to the same state. Clients issue requests to the servers using an atomic multicast pro-

To appear in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, November 1994.

tocol, so that all honest servers process requests in the same order and thus produce the same output for each request. If the honest servers sufficiently outnumber the corrupt ones, then a client can identify the correct output by using a simple voting scheme, thereby masking out the effects of corrupt servers. Note that this approach *requires* atomic multicast in general. Thus, our atomic multicast protocol, and the reliable multicast protocol on which it is built, are central to Rampart.

To our knowledge, Rampart is the first system to demonstrate the feasibility of reliable and atomic group multicast tolerant of malicious processes in loosely coupled distributed systems. Prior systems that demonstrated reliable or atomic multicast tolerant of malicious processes required a synchronous network for correctness, in which there are known bounds on message transmission times, process execution rates, and relative clock drifts [7, 19, 29]. Due to these synchrony assumptions, this work is inappropriate for loosely coupled systems, and especially for hostile settings where messages may be delayed due to denial-of-service attacks [32].

Our protocols rely on no synchrony assumptions, even though it is impossible to (deterministically) solve atomic multicast without them [8]. We circumvent this impossibility result with a method similar to that used in some systems that provide atomic multicast tolerant of benign failures (e.g., [3, 1]). Specifically, we build our group multicast protocols over a *secure group membership protocol* [21] that enables honest group members to collectively remove unresponsive members from the group during a multicast, so the honest members can make progress. This carries the risk of removing an honest but unreachable member from the group (and the multicast protocol). But it also removes the theoretical barrier that prevents atomic multicast from being solved—i.e., that it is impossible to tell if a member has failed or is only unreachable—by deeming unresponsive members corrupt and removing them from the protocol. A removed process must rejoin the group via the membership protocol before it can take part in further multicast protocols. Our protocols (and the protocol of [21]) require that more than two-thirds of the members in each instance of the group membership are honest.

In our present implementation, our protocols offer performance that suffices for many applications, including those that initially motivated Rampart. However, they also indicate a high cost for tolerating malicious processes. In fact, the latencies of our protocols in our present implementation exceed those of some published protocols for benign failures by an order of magnitude; similar results hold for multicast throughput. Most of our protocol cost is computational due to the cryptographic operations performed, and exploiting special-purpose hardware for these operations would greatly improve their performance.

The rest of this paper proceeds as follows. In Section 2 we state our assumptions about the system. In Section 3, we detail the semantics of our multicast protocols. In Section 4, we present a protocol that we use to implement our multicast protocols, and Sections 5 and 6 present the multicast protocols themselves. Section 7 explores a way to strengthen the semantics of our protocols. Section 8 discusses our protocols' performance in our present implementation. Appendix A proves some properties of our protocols, although for brevity, we defer most proofs to a forthcoming extended paper.

2 System model

We assume a system consisting of some number of *processes* $p_1, p_2, p_3, \ldots$ We will often denote processes with the letters p, q and r when subscripts are unnecessary. A process that behaves according to its specification is said to be *honest.* A *corrupt* process, however, can behave in any fashion whatsoever, limited only by the assumptions stated below. Corrupt processes include those that fail benignly.

Processes communicate via a network that provides a FIFO point-to-point communication channel between each pair of processes. These channels are authenticated and protect the integrity of communication using, e.g., well-known cryptographic techniques [32]. To show liveness of our protocols, we assume that communication is reliable, in the sense that if the sender and destination of a message are honest, then the destination eventually receives the message. However, we *do not* assume known, finite upper bounds on message transmission times; i.e., communication is asynchronous. Moreover, we *do not* assume that processes maintain synchronized clocks or clock rates. While our protocols do use timeouts to ensure progress, these uses do not require clock synchronization among processes.

Each process p_i possesses a private key K_i known only to itself, with which it can digitally sign messages using a digital signature scheme (e.g., [26]). We denote a message $\langle \ldots \rangle$ signed with K_i by $\langle \ldots \rangle_{K_i}$. 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.[1]

As discussed in Section 1, we assume the existence of a *group membership protocol* that provides a process group abstraction to processes [21]. This protocol gener-

[1] This assumption and the previous assumption of secure channels may seem to conflict with a motivation discussed in Section 1, namely that Rampart will be used to build services that support cryptographic key distribution. In this case, our assumptions equate to only a requirement that the cryptographic key *servers* can communicate securely among themselves and have access to each others' public keys, e.g., by a "manual" administrative action. The service will then distribute keys to a larger system with the availability and integrity guarantees outlined in Section 1.

ates a sequence $V^1, V^2, \ldots$ of *group views*, each of which is a set of process identifiers. Each V^{x+1} differs from V^x by the addition or removal of a single process. Our protocols (and the protocol of [21]) require that at most $\lfloor (|V^x| - 1)/3 \rfloor$ members of each V^x are corrupt, and thus that at least $\lceil (2|V^x| + 1)/3 \rceil$ members of each V^x are honest. The membership protocol ensures that each honest p receives V^x iff $p \in V^x$, and that p receives V^x before V^y if $x < y$ (and p is in both). The membership protocol also supports interfaces by which processes can influence future membership changes. For this paper, two of these interfaces are important: a member of V^x can execute *remove*(x, p) to request that some $p \in V^x$ be removed from the group to form V^{x+1}, and it can execute *adds*(x) to enable additions to occur in view x. These operations have the following semantics:

- If $p \in V^x$ and for all k such that $p \in \bigcap_{0 \le k' \le k} V^{x+k'}$, at least $\lfloor (|V^{x+k}| - 1)/3 \rfloor + 1$ honest members of V^{x+k} execute *remove*$(x + k, p)$, then there exists some V^y, $y > x$, such that $p \notin V^y$.
- If no honest member of V^x executes *adds*(x) prior to the generation of V^{x+1}, then $V^{x+1} \subseteq V^x$.

Intuitively, the first property says that if enough honest members of each V^{x+k}, $k \ge 0$, request to remove p from the group, then p is eventually removed. For simplicity, we assume that if an honest $q \in V^x$ executes *remove*(x, p), then it also executes *remove*$(x + k, p)$ for each V^{x+k} it receives such that $\{p, q\} \subseteq \bigcap_{0 \le k' \le k} V^{x+k'}$; i.e., q executes *remove*$(x + k, p)$ until either it or p is removed. Even with this, however, to remove p it may not suffice for all honest members of V^x to execute *remove*(x, p), because if processes are then added to the group, those processes that requested p's removal may no longer constitute enough of the group to remove p. This is precisely the motivation for the second property above, which provides a way for honest members to prevent additions to the group until a member can be removed. We will see how this is used in Section 5 and Appendix A.

These properties are somewhat different from those of the membership protocol described in [21]. However, they can be achieved (or effectively approximated in practice) with minor modifications to that protocol.

3 Multicast semantics

In this section we more carefully state the semantics of our reliable and atomic group multicast protocols. Our reliable multicast protocol provides an interface *R-mcast*(m), by which a process can multicast a message m to the group. A process delivers a message m from p via the reliable multicast protocol by executing *R-deliver*(p, m). In addition, if a process receives the x-th group view V^x, it can deliver that view via the reliable multicast protocol by executing *R-deliver*(V^x). R-mcasts and R-deliveries at an honest process occur strictly sequentially, and group views are R-delivered in order of increasing x. An execution of *R-mcast* or *R-deliver* at a process p is said to occur *in* view x if the last view R-delivered at p prior to that execution is V^x. It is convenient to assume that an honest process does not R-mcast the same message twice in the same view; this can be enforced, e.g., by the process including a sequence number in each message.

As described in Section 1, the task of a reliable multicast protocol is to ensure that group members deliver the same messages. We capture this semantic in four properties: Integrity, Agreement, Validity-1, and Validity-2. As stated below, Agreement, Validity-1, and Validity-2 restrict behavior only at honest processes that are not removed from the group after some point (in practice, for sufficiently long). We can strengthen these properties in various ways; Section 7 covers one alternative. For now, however, we content ourselves with the properties below.

Integrity: For all p and m, an honest process executes *R-deliver*(p, m) at most once in view x and, if p is honest, only if p executed *R-mcast*(m) in view x.

Agreement: If p and q are honest members of V^{x+k} for all $k \ge 0$ and p executes *R-deliver*(r, m) in view x, then q executes *R-deliver*(r, m) in view x.

Validity-1: If p is an honest member of V^{x+k} for all $k \ge 0$, then p executes *R-deliver*(V^x).

Validity-2: If p and q are honest members of V^{x+k} for all $k \ge 0$ and p executes *R-mcast*(m) in view x, then q executes *R-deliver*(p, m) in view x.

Agreement roughly captures our intuitive definition of reliable multicast—i.e., that honest members deliver the same messages—but Integrity is also needed to meet this description. Validity-1,2 rule out trivial solutions by requiring that views and messages be R-delivered. Validity-2 is noteworthy as it ensures the R-delivery of messages from honest *members* only. In fact, as presented here our protocol does not allow multicasts from outside the group (i.e., groups are *closed* [13]), but it easily extends to allow such interactions.

The specification of atomic multicast includes Integrity, Agreement, Validity-1, and Validity-2, and adds a property that ensures that messages are delivered by group members in the same order. More precisely, our atomic multicast protocol provides an interface *A-mcast*(m), by which a process can multicast a message m to the group. A process delivers a message m from p via the atomic multicast protocol by executing *A-deliver*(p, m). As before, a process can also deliver a view V^x via the atomic multicast protocol by executing *A-deliver*(V^x), and an execution of *A-mcast* or *A-deliver* at a process p is said to occur *in* view x if the last view A-delivered at p prior to that execution is view x. We again assume that an honest process does not A-mcast the same message twice in the same view. Our atomic multicast protocol A-delivers messages to group members according to the following semantics. First, Integrity, Agreement, Validity-1, and Validity-2 hold (with *R-mcast* and *R-deliver* replaced with *A-mcast* and *A-deliver*, respectively). Second, atomic multicast provides the following additional property:

Order: If p and q are honest members of V^{x+k} for all $k \geq 0$ and p executes *A-deliver*(r, m) before *A-deliver*(r', m') in view x, then q executes *A-deliver*(r, m) before *A-deliver*(r', m') in view x.

A consequence of stating these properties (except Integrity) in a way that restricts behavior only at honest processes that are not removed from the group is that for these properties to be meaningful in practice, corrupt processes must not be able to easily cause the removal of honest members from the group. Our protocols (and the membership protocol of [21]) are designed to ensure that corrupt members cannot cause the removal of honest members except by performing network denial-of-service attacks that delay messages sent between honest members to the extent that some honest members appear unreachable. For brevity, we omit further discussion of these issues here.

4 Echo multicast

A core component of our reliable and atomic multicast protocols is a protocol called *echo multicast*. In fact, in the absence of membership changes, a reliable multicast essentially reduces to a single echo multicast. Our echo multicast protocol provides an interface *E-mcast*(x, m) by which a process in V^x can multicast a message m to the members of V^x. A process delivers a message m for view x from $p \in V^x$ via the echo multicast protocol by executing *E-deliver*(p, x, m). The echo multicast protocol ensures that the l-th E-deliveries from p for view x at any two honest processes are the same.

4.1 The basic protocol

Echo multicast is perhaps best understood in a simplified form that enables a single $p \in V^x$ to multicast only a single message m to V^x. To initiate the multicast, p sends m to all members of V^x. When a member receives a message m from p, it "echoes" m by digitally signing m and returning this to p; additional messages received from p are not echoed. Once

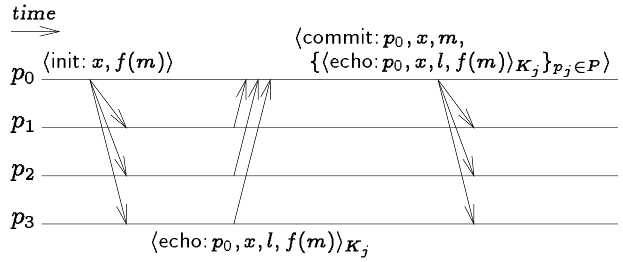

Figure 1: Echo multicast

p receives $\lceil (2|V^x| + 1)/3 \rceil$ echoes for m, it sends these echoes to all members of V^x. When a process receives these $\lceil (2|V^x| + 1)/3 \rceil$ echoes for m and verifies their signatures, it E-delivers m. The number $\lceil (2|V^x| + 1)/3 \rceil$ is significant because if at most $\lfloor (|V^x| - 1)/3 \rfloor$ members of V^x are corrupt, it ensures that a majority of the honest members of V^x echoed m. Since an honest member echoes only one message from p, honest members cannot E-deliver different messages, even if p is corrupt.

To generalize this protocol to handle many multicasts from different processes, each echo is modified to include the name of the multicast initiator, the view number, and a sequence number in its signed contents. The protocol also employs a *message digest* function to reduce the number of times that the message m is transmitted on the network. A message digest function f maps any arbitrary length input m to a fixed length output $f(m)$ and has the property that it is computationally infeasible to determine two inputs m and m' such that $f(m) = f(m')$. Several efficient message digest functions have been proposed; the Rampart toolkit currently offers MD4 [24] and MD5 [25].

The echo multicast protocol then executes as follows. Each $p \in V^x$ maintains a set of counters $\{c_i^x\}_{p_i \in V^x}$, each initially zero, and a set *commits*^{x} of messages, which is initially empty. Each counter c_i^x keeps track of the number of messages that have been E-delivered for view x from p_i, and is used to E-deliver messages from p_i in FIFO order. The steps of the protocol are listed below (see Figure 1).

1. If *E-mcast*(x, m) is executed at some $p \in V^x$, p sends

$$\langle \mathsf{init}: x, f(m)\rangle$$

to each member of V^x. (A message sent by a process to itself is received immediately when it is sent.) This process is called the multicast *initiator*.

2. If p_j receives $\langle \mathsf{init}: x, d\rangle$ from some $p \in V^x$ and this is the l-th message of the form $\langle \mathsf{init}: x, *\rangle$[2] that p_j has

[2] As a notational convenience, throughout this paper we use "$*$" to mean "anything", i.e., a wild-card value.

received from p, then p_j sends

$$\langle \mathsf{echo}\colon p, x, l, d\rangle_{K_j}$$

to p.

3. Once the initiator p has received a set of echoes $\{\langle \mathsf{echo}\colon p, x, l, f(m)\rangle_{K_j}\}_{p_j \in P}$ for some l and some $P \subseteq V^x$ where $|P| = \lceil (2|V^x|+1)/3\rceil$, it sends

$$\langle \mathsf{commit}\colon p, x, m, \{\langle \mathsf{echo}\colon p, x, l, f(m)\rangle_{K_j}\}_{p_j \in P}\rangle$$

to each member of V^x.

4. If a process receives

$$\langle \mathsf{commit}\colon p_i, x, m, \{\langle \mathsf{echo}\colon p_i, x, l, f(m)\rangle_{K_j}\}_{p_j \in P}\rangle$$

for some $l > c_i^x$ and some $P \subseteq V^x$ where $|P| = \lceil (2|V^x|+1)/3\rceil$, and if it has not received a view V^y, $y > x$, such that $p_i \notin V^y$, then it adds this commit message to *commits*^{x}.

5. Whenever a process adds a message $\langle \mathsf{commit}\colon p_i, \ldots\rangle$ to *commits*^{x}, it repeats the following step until it results in no more E-deliveries: if there is a message

$$\langle \mathsf{commit}\colon p_i, x, m, \{\langle \mathsf{echo}\colon p_i, x, l, f(m)\rangle_{K_j}\}_{p_j \in P}\rangle \quad (1)$$

in *commits*^{x} such that $c_i^x + 1 = l$, then it executes *E-deliver*(p_i, x, m) and sets $c_i^x \leftarrow c_i^x + 1$.

Two points are worth noting. First, by step 4, a process ignores any $\langle \mathsf{commit}\colon p, x, \ldots\rangle$ that it receives on the network if p has been removed from the group (i.e., if some V^y, $y > x$, has been received such that $p \notin V^y$). This is needed for the correctness of our reliable multicast protocol, discussed in Section 5. Second, for a process to execute this protocol for view x, it must have received V^x. So, if a process receives a protocol message for a view that it has not yet received, it buffers the message until it receives that view.

The protocol described above is similar to the *echo broadcast* protocol presented in [30], although our protocol scales better as a function of $|V^x|$ in the number of messages and transmissions of m (at the cost of greater computation and stronger cryptographic assumptions). Specifically, an execution of *E-mcast*(x, m) results in $O(|V^x|)$ messages and transmissions of m, versus $O(|V^x|^2)$ with the protocol of [30].

Under the assumption that at most $\lfloor (|V^x| - 1)/3\rfloor$ members of V^x are corrupt, the following hold.

Lemma 1 *If p is honest and some honest process executes E-deliver(p, x, m), then p executed E-mcast(x, m).*

Lemma 2 *If the l-th executions of E-deliver$(p, x, *)$ at two honest processes are E-deliver(p, x, m) and E-deliver(p, x, m'), respectively, then $m = m'$.*

Therefore, this protocol protects the integrity of E-mcasts, and it ensures that honest processes do not disagree on the contents of an echo multicast.

4.2 Stability

We say that the l-th echo multicast from p_i to view x is *stable* if $c_i^x \geq l$ at every honest member of V^x, or in other words, if the l-th E-delivery of the form *E-deliver*$(p_i, x, *)$ has been executed at every honest member of V^x. For our reliable multicast protocol, it is necessary (in the absence of membership changes) for processes to retain the commit message for each echo multicast until the multicast is stable. That is, a process removes message (1) from *commits*^{x} only after it determines that $c_i^x \geq l$ at all honest members of V^x.

So that processes can tell when echo multicasts are stable, each $q \in V^x$ periodically echo multicasts its set $\{c_i^x\}_{p_i \in V^x}$ of counter values to view x. q can piggyback this set on another E-mcast to view x, or if q is not already E-mcasting a message to view x, it can E-mcast its counter values to V^x in a separate message. A process q knows that the l-th echo multicast from p_i to view x is stable if q has E-delivered a message for view x containing a counter value $c_i^x \geq l$ from each member of V^x.

An honest process does not permit a multicast to remain unstable for longer than a prespecified timeout duration. That is, if a process q retains (1) in *commits*^{x} beyond some timeout duration after executing *E-deliver*(p_i, x, m), it attempts to make this multicast stable by sending (1) to each $r \in V^x$ from which q has not E-delivered a counter value $c_i^x \geq l$. If within some timeout duration after sending (1) to r, q does not receive V^{x+1} or E-deliver a counter value $c_i^x \geq l$ from r, then q executes *remove*(x, r). The purpose of this request is to prevent a corrupt r from causing honest members of V^x to retain (1) indefinitely, which could be costly in terms of storage if m or V^x is large. If all honest members of V^x (and thus at least $\lceil (2|V^x|+1)/3\rceil \geq \lfloor (|V^x| - 1)/3\rfloor + 1$ honest members of V^x) execute *remove*(x, r), then V^{x+1} must eventually be generated, due to the semantics of *remove* (see Section 2). As we discuss in Section 5, if q receives V^{x+1} (whether or not $r \in V^{x+1}$), then it will eventually be able to discard all messages in *commits*^{x} after R-delivering V^{x+1}. If V^{x+1} is not generated, then some honest member of V^x must have E-delivered a message m' containing the required counters from r. As described above, this member will forward $\langle \mathsf{commit}\colon r, x, m', \ldots\rangle$ to q if necessary, thereby allowing q to discard (1).

5 Reliable multicast

In this section we describe the reliable multicast protocol of Rampart. When there are no membership changes, a reliable multicast is implemented by a single echo multicast. That is, if a process executes *R-mcast*(m) in view x, then it implements this by executing *E-mcast*$(x, \langle \mathsf{r\text{-}msg}\colon m\rangle)$. If a process executes

E-deliver$(p,\ x,\ \langle\mathsf{r\text{-}msg}{:}\,m\rangle)$ and view x was the last view R-delivered, then it executes *R-deliver*(p, m).

The protocol becomes somewhat more complex during membership changes. First suppose that only a single membership change occurs. When the new view V^{x+1} is received at $p \in V^x \cap V^{x+1}$, p inhibits new reliable multicasts and E-mcasts an end message to V^x. This end message marks the end of reliable multicasts from p in view x. Once p has E-delivered an end message for view x from every member of $V^x \cap V^{x+1}$, it executes *E-mcast*$(x+1,\ \langle\mathsf{flush}{:}\, \mathit{commits}^x\rangle)$. The purpose of this flush message is to communicate to the members of $V^x \cap V^{x+1}$ the commit messages for view x that p received. When each process in $V^x \cap V^{x+1}$ E-delivers this flush message from p, it also adds the (properly formed) commit messages in p's flush to its own $\mathit{commits}^x$ set (and E-delivers messages as appropriate). Once p has E-delivered a flush from every member of V^{x+1}, it R-delivers V^{x+1} and resumes reliable multicasts.

As described, this protocol suffices only for a single membership change. However, membership changes might occur during the execution of this protocol, and in fact it may sometimes be necessary to force membership changes to occur so that this protocol can make progress. In particular, if an end or flush message is never E-delivered from a process, it will be necessary to remove that process from the group in order to make progress. To make such removals possible, when an honest process receives V^{x+1}, it does not enable additions in V^{x+1} until it R-delivers V^{x+1}. While additions are disabled, honest members can remove unresponsive members as needed, thereby generating new views. When each new view is received, the same protocol is executed for the new view. This continues until some V^{x+k} is received such that flush messages for $V^{x+1}\ldots V^{x+k}$ are E-delivered from every member of $\bigcap_{0<k'\le k} V^{x+k'}$, i.e., from every member of V^{x+1} that is still in the group. Only then can V^{x+1} be R-delivered.

We now state the protocol more carefully. Each $p \in V^x$ maintains a FIFO queue defer^x, initially empty, which is used to defer R-deliveries intended for view x until V^x is R-delivered. An honest process does not execute *R-mcast* in view x if it has received a view V^y where $y > x$. The reliable multicast protocol at each process in V^x executes as follows.

1. If *R-mcast*(m) is executed in view x, then execute *E-mcast*$(x,\ \langle\mathsf{r\text{-}msg}{:}\,m\rangle)$.

2. If *E-deliver*$(p,\ x,\ \langle\mathsf{r\text{-}msg}{:}\,m\rangle)$ is executed and neither *E-deliver*$(p,\ x,\ \langle\mathsf{end}\rangle)$ nor *E-deliver*$(p,\ x,\ \langle\mathsf{r\text{-}msg}{:}\,m\rangle)$[3] was previously executed:
   a. If V^x is the most recently R-delivered view, then execute *R-deliver*(p, m).
   b. If V^x has not yet been R-delivered, then enqueue the pair (p, m) on defer^x.

3. If view V^{x+1} is received:
   a. Execute *E-mcast*$(x,\ \langle\mathsf{end}\rangle)$.
   b. For each $p \in V^x \cap V^{x+1}$, if *E-deliver*$(p,\ x,\ \langle\mathsf{end}\rangle)$ is not executed within some predetermined timeout duration, then execute *remove*$(x+1,\ p)$.

4. When for some $k > 0$, views $V^{x+1}\ldots V^{x+k}$ have been received and for all $p \in \bigcap_{0\le k'\le k} V^{x+k'}$, *E-deliver*$(p,\ x,\ \langle\mathsf{end}\rangle)$ and *E-deliver*$(p,\ \overline{x},\ \overline{\langle}\mathsf{flush}{:}\ *\rangle)$ have been executed:
   a. Execute *E-mcast*$(x+1,\ \langle\mathsf{flush}{:}\,\mathit{commits}^x\rangle)$.
   b. For each $p \in V^{x+1}$, if *E-deliver*$(p,\ x+1,\ \langle\mathsf{flush}{:}\ *\rangle)$ is not executed within some predetermined timeout duration, then execute *remove*$(x+1,\ p)$.

5. If V^x is the most recently R-delivered view:
   a. If *E-deliver*$(p, x+k, \langle\mathsf{flush}{:}\,S\rangle)$ is (or was priorly) executed where $p \in \bigcap_{0\le k'\le k} V^{x+k'}$, $k > 0$ and $V^{x+1}\ldots V^{x+k}$ have been received, and if this is the first execution of *E-deliver*$(p, x+k, \langle\mathsf{flush}{:}\ *\rangle)$, then for $y = x+k-1$, add to $\mathit{commits}^y$ each message

   $$\langle\mathsf{commit}{:} p_i, y, m, \{\langle\mathsf{echo}{:}\, p_i, y, l, f(m)\rangle_{K_j}\}_{p_j\in P}\rangle$$

   in S such that $l > c_i^y$, $P \subseteq V^y$, and $|P| = \lceil(2|V^y|+1)/3\rceil$. (Also apply rule 5 of the echo multicast protocol of Section 4.1.)
   b. When for some $k > 0$, views $V^{x+1}\ldots V^{x+k}$ have been received and *E-deliver*$(p,\ x+k',\ \langle\mathsf{flush}{:}\ *\rangle)$ has been executed for all $p \in \bigcap_{0<k''\le k} V^{x+k''}$ and all k', $0 < k' \le k$:
      i. Execute *R-deliver*(V^{x+1}) and *adds*$(x+1)$.
      ii. Repeat the following until defer^{x+1} is empty: dequeue the head of defer^{x+1}, say (q, m), and execute *R-deliver*$(q,\ m)$.

A process in $V^{x+1} - V^x$ (or V^{x+1} if $x = 0$) also executes parts of this protocol, beginning at step 4a. More precisely, when the process receives V^{x+1}, it executes *E-mcast*$(x+1,\ \langle\mathsf{flush}{:}\,\emptyset\rangle)$ and, if $x = 0$, step 4b. It then executes step 5b (ignoring the condition of step 5 that V^x be R-delivered), and the full protocol for V^{x+1}.

Two points about step 5 deserve clarification. First, due to step 5a, the E-delivery of a flush for view $x+k$ could result in the E-delivery of a flush for view $x+k-1$, which could result in the E-delivery of a flush for view $x+k-2$, etc. Thus, the E-delivery of a flush for view $x+k$ could "cascade" into an E-delivery for view x and thus an R-delivery in view x. If a flush for view $x+k$

[3] In practice, duplicates can be detected by a sequence number in m, rather than based on m's entire contents. These sequence numbers need only be unique among messages R-mcast by the same process in the same view.

is E-delivered prior to the R-delivery of V^{x+1}, then it is essential that any E-deliveries that could result by step 5a be performed prior to R-delivering V^{x+1} in step 5b. Second, note that if V^x is the most recently R-delivered view, then step 5a considers *E-deliver*(p, $x+k$, $\langle$flush: $*\rangle$) only if $p \in \bigcap_{0\le k'\le k} V^{x+k'}$. Once V^{x+1} is R-delivered, prior E-deliveries of flush messages from any $p \in V^{x+1} - V^x$ should be revisited to determine if they now satisfy the conditions of 5a in the protocol for V^{x+1}.

Once a process p R-delivers V^{x+1}, it participates in no new echo multicasts for view y and adds no more commit messages to $commits^y$ for any $y \le x$. It then empties $commits^y$ for all $y \le x$ after sending each $\langle \text{commit}: q, y, m \ldots\rangle \in commits^y$ to each $r \in V^y$ that did not execute *E-deliver*(q, y, m), according to the counters that p E-delivered for view y from r (see Section 4.2).

The proofs of Integrity and Validity-2 for this protocol are straightforward and are omitted here for the sake of brevity. The proofs for Validity-1 and Agreement, however, are less straightforward and are sketched in Appendix A. Finally, we state without proof a lemma that is necessary for the correctness of our atomic multicast protocol of Section 6.

Lemma 3 *If p and q are honest members of V^{x+k} for all $k \ge 0$ and p executes R-deliver(r,m) before R-deliver(r,m') in view x, then q executes R-deliver(r,m) before R-deliver(r,m') in view x.*

Intuitively, this lemma states that honest processes R-deliver messages from the same process in the same order. Note that this differs from the Order property of Section 3 in two ways: as stated, it applies only to R-deliveries (not A-deliveries), and then only to the R-delivery order of messages *from the same process.* This lemma follows easily from Lemma 2.

6 Atomic multicast

Our atomic multicast protocol is built over the reliable multicast protocol of Section 5. The semantics of reliable multicast greatly simplify our atomic multicast protocol. In fact, they enable us to use a protocol that is similar to those used in the Amoeba [13] and Isis [3] systems, which are tolerant only to benign failures. Intuitively, the protocol works by allowing a designated member of each group view, called the *sequencer*, to determine the order in which messages in that view are A-delivered. The sequencer in view x can be determined using any deterministic algorithm (e.g., the member of V^x with the lexicographically smallest identifier), because the x-th view at each honest $p \in V^x$ is the same.

Informally, the protocol executes as follows. To A-mcast a message, a process simply R-mcasts it. As the sequencer R-delivers messages, it chooses an A-delivery order for the messages and periodically R-mcasts a special order message indicating this chosen A-delivery order. When a process R-delivers an order message from the sequencer, it A-delivers messages in the order indicated in the order message. If a new view V^{x+1} is R-delivered, there may be some messages that were R-delivered in view x but for which no order message was R-delivered in view x that indicates the order in which these messages should be A-delivered. However, since this set of non-A-delivered messages is guaranteed to be the same at all honest members that are not removed from the group, the members can A-deliver these messages according to any deterministic ordering (e.g., ordered by the processes that R-mcast them, and by R-delivery order among messages R-mcast by the same process). Once all (non-order) messages R-delivered in view x are A-delivered, V^{x+1} can be A-delivered.

While Integrity, Agreement, and Validity-1 (with *R-mcast*/*R-deliver* replaced by *A-mcast*/*A-deliver*), as well as Order, are achieved by this protocol, additional steps are needed to ensure that a corrupt sequencer cannot prevent the A-delivery of a message and thus cause a violation of Validity-2. A corrupt sequencer could try to prevent the A-delivery of a message either by refusing to report when to A-deliver it, or by including messages before it in the chosen A-delivery order that will never be R-delivered. To counter such behaviors, if an honest process p does not A-deliver a message within some predetermined timeout period after R-delivering it, then p requests that the sequencer be removed from the group. This request is justified because any message R-delivered by the sequencer should soon be R-delivered by p as well, and vice versa, provided that the sequencer is honest and reachable (see Section 4.2). Once p R-delivers a new view, it can A-deliver any non-A-delivered messages as before, i.e., according to some deterministic ordering. If the membership protocol does not generate a new view, then by Agreement for R-deliveries, p must eventually R-deliver the messages on which it is waiting, i.e., the order message placing the delayed message in the A-delivery sequence or the messages to be A-delivered prior to the delayed message.

We now state the protocol more carefully. Each $q \in V^x$ maintains a set of FIFO queues $\{pending^x_i\}_{p_i \in V^x}$, each of which is initially empty, and a FIFO queue $order^x$ of process identifiers, initially empty. The $pending^x_i$ queues are used to store R-delivered messages that are awaiting A-delivery, and $order^x$ is used to record the A-delivery order chosen by the sequencer. A dequeue operation on an empty queue blocks until an enqueue operation is executed on the queue. The sequencer for V^x, denoted seq^x below, maintains a list $senders^x$ of process identifiers, initially empty. An honest process does not execute *A-mcast* in view x if it has received a view V^y where $y > x$. The protocol for V^x, which is initiated at each $q \in V^x$ when q A-delivers V^x, executes as follows.

1. If *A-mcast*(m) is executed in view x, then execute *R-mcast*($\langle$a-msg: $m\rangle$) (in view x).

2. If *R-deliver*(p_i, $\langle$a-msg: $m\rangle$) is executed in view x:

   a. Enqueue m on $pending_i^x$.

   b. (seq^x only) Set $senders^x \leftarrow senders^x || p_i$, where $||$ denotes concatenation.

   c. If *A-deliver*(p_i, m) is not executed within some prespecified timeout period, execute *remove*(x, seq^x).

3. (seq^x only) Periodically do the following until some V^y, $y > x$, is received:

   a. Execute *R-mcast*($\langle$order: $senders^x\rangle$) (in view x).

   b. Set $senders^x \leftarrow \emptyset$ (i.e., the empty list).

4. If *R-deliver*(seq^x, $\langle$order: $p_{i_1} \ldots p_{i_n}\rangle$) is executed in view x, then for $j = 1 \ldots n$ (in increasing order): if $p_{i_j} \in V^x$ then enqueue p_{i_j} on $order^x$.

5. Repeat the following until some V^y, $y > x$, is A-delivered:

   a. Dequeue the head of $order^x$, say p_i.

   b. Dequeue the head of $pending_i^x$, say m.

   c. Execute *A-deliver*(p_i, m).

6. If *R-deliver*(V^{x+1}) is executed:

   a. Let $p_{i_1} \ldots p_{i_n}$ ($n = |V^x|$) be a deterministic enumeration of V^x (e.g., $i_j < i_{j+1}$ for all j).

   b. For $j = 1 \ldots n$ (in increasing order), repeat the following while $pending_{i_j}^x$ is not empty: dequeue the head of $pending_{i_j}^x$, say m, and execute *A-deliver*(p_{i_j}, m).

   c. Execute *A-deliver*(V^{x+1}).

It is important that as many messages as possible be A-delivered via step 5—i.e., according to the order described in the $order^x$ queue—before A-delivering messages according to step 6. Also, it is worth clarifying that while steps 1, 2, 4, and 6 are executed in response to certain events, steps 3 and 5 are continuously executed, beginning when the process A-delivers V^x. A process in $V^{x+1} - V^x$ (or V^{x+1} if $x = 0$) A-delivers V^{x+1} immediately upon R-delivering it.

In Rampart, we apply several optimizations to this protocol. In particular, whenever possible the sequencer piggybacks ordering information on the a-msg messages that it R-mcasts, so that this information need not be R-mcast separately. The impact of this optimization is discussed in Section 8.

7 Uniformity

As discussed in Section 3, the Agreement and Order properties restrict behavior at honest processes that are not removed from the group. However, these properties say nothing about honest members that *are* removed from the group. For example, with the protocol of Section 5, it is possible for an honest member of $V^x - V^{x+1}$ to R-deliver messages in view x that members of $\bigcap_{k \geq 0} V^{x+k}$ do not. Similarly, with the protocol of Section 6, it is possible for an honest member of $V^x - V^{x+1}$ to A-deliver messages in view x in a different order than members of $\bigcap_{k \geq 0} V^{x+k}$.

There exist applications for which such possibilities may be problematic. For example, consider a distributed database that implements a *Chinese wall* [4] access control policy on data sets. Briefly, a Chinese wall policy groups data sets into "conflict-of-interest classes", and each subject is allowed to access at most one data set belonging to each class. To prevent a client from gaining access to different data sets in the same conflict-of-interest class from different data servers, one approach would be to issue all requests to the database servers with an atomic multicast protocol. Intuitively, since all honest servers would deliver access requests in the same order, for each request all honest servers would agree on the sequence of prior requests from the same client and thus would agree on whether to grant the request. However, this would not necessarily be the case if our atomic multicast protocol of Section 6 were used: if an honest server were partitioned away from the rest of the group at an inopportune moment and then removed, two "conflicting" requests could be A-delivered at this honest server in the opposite order from other honest servers, possibly resulting in a policy violation.

Such problems can be addressed by strengthening Agreement and Order to prevent removed but honest processes from taking actions that honest group members do not; such properties are said to be *uniform* [27]. In the case of reliable multicast, one such strengthening can be stated as follows.

Uniform Agreement: If q is an honest member of V^{x+k} for all $k \geq 0$ and an honest p executes *R-deliver*(r, m) in view x, then q executes *R-deliver*(r, m) in view x.

Note that this strengthens Agreement by relaxing the requirement that $p \in V^{x+k}$ for all $k \geq 0$. Stronger uniformity requirements are possible (cf. [27]). Uniform Agreement can be achieved with minor changes to our reliable multicast protocol, similar to the methods described in [1, 27] for benign failures. Very briefly, each process defers the R-delivery of a message m in view x until for some $k \geq 0$, $\langle$r-msg: $m\rangle$ has been E-delivered at all honest members of $\bigcap_{0 \leq k' \leq k} V^{x+k'}$. In the absence of membership changes, this means that m is not R-delivered until the echo multicast of $\langle \mathsf{r\text{-}msg}: m\rangle$ is stable. For brevity, we omit further discussion of this protocol.

Uniform Agreement for atomic multicast can be formulated by replacing *R-deliver* with *A-deliver* in the above statement. To ensure consistent behavior of a removed process in the case of atomic multicast, however, Order also must be strengthened.

Uniform Order: If q is an honest member of V^{x+k} for all $k \geq 0$ and an honest p executes *A-deliver*(r, m) before *A-deliver*(r', m') in view x, then q executes *A-deliver*(r, m) before *A-deliver*(r', m') in view x.

The atomic multicast protocol of Section 6 satisfies both Uniform Agreement and Uniform Order if built over the reliable multicast protocol mentioned above that satisfies Uniform Agreement. Intuitively, this is because an honest $q \in \bigcap_{k\geq 0} V^{x+k}$ R-delivers all messages in view x that an honest $p \in V^x$ does (by Uniform Agreement for reliable multicast), and if p R-delivers V^{x+1}, then p R-delivers all messages in view x that q does (see Lemma 6 of Appendix A). So, any A-deliveries at p—which must result from its R-delivery of order messages or V^{x+1}—will occur in the same order as A-deliveries at q.

Despite the simplicity of the additional measures needed to satisfy Uniform Agreement and Uniform Order in our protocols, these measures can significantly impact the latency of multicasts, and thus they are not currently supported in the Rampart toolkit. Instead, we implement a functionally similar, but more efficient and flexible, mechanism that delays not the *delivery* of messages, but rather the "visibility" of any actions that result from those deliveries. That is, if the application so requests, Rampart delays sending messages that resulted from the A-delivery of a message in view x until that message is sure to be A-delivered in the same relative order at all honest members that are not removed from the group. To accommodate applications that take external actions based on messages A-delivered to it, we are experimenting with mechanisms to inform the application when it is safe to take action.

8 Performance

Experience with current group-oriented systems suggests that group membership changes are infrequent for most applications (see [3]), and we expect this to be the case in the Rampart applications that we currently envision. Thus, we expect that the most frequently observed costs associated with our multicast protocols will be those that occur in the absence of membership changes. In this section, we discuss these costs when there are no process corruptions, using data gathered from our prototype implementation of Rampart. This implementation employs CryptoLib [14] for its cryptographic operations and runs over the Multicast Transport Service [31], which supports point-to-point authenticated channels [23]. The public key cryptosystem we use is RSA [26].

As reliable and atomic multicasts are implemented essentially as one or two echo multicasts in the absence of membership changes, it is crucial that echo multicast perform well. The factor dominating the cost of an echo multicast in our present implementation is the cost of RSA operations: each group member must perform a digital signature and verify multiple signatures per echo multicast. In order to minimize the cost of signature verifications for a given RSA modulus size, we set all public exponents equal to three in our system. This is a common optimization in practice, and is reasonable for use in our system because these exponents are used only to verify signatures, and never to encrypt messages. (For weaknesses resulting from using small exponents for encryption, see [18].) However, this optimization has no effect on the cost of creating signatures.

To optimize signature creation, each process uses a different short-term public key pair in the echo multicast protocol for each view. Briefly, in its first echo multicast to a view, each process piggybacks a new, short-term public key on the message being multicast. Until this multicast is stable, the process p_i uses its long-term private key K_i for signing echoes for that view. However, once the multicast is stable, it switches to using the short-term private key corresponding to the public key that it echo multicast. To limit the duration for which short-term keys are used, the membership protocol periodically generates a null view change (i.e., a new view V^{x+1} identical to V^x), thereby causing new short-term keys to be deployed. Here we omit further details of how short-term keys are used, except to note that their use does require slight alterations to our reliable multicast protocol to ensure its correctness. Since short-term keys can be computed in the background, and since the same short-term key can be used in several views received in quick succession, the cost of generating short-term keys has no perceivable impact on our protocols.

The benefit of using short-term keys is that the size of these keys can be minimized according to the maximum time they will be in use. For instance, if the membership protocol generates a new view at least once per hour, then the RSA modulus of a short-term key can be set to a length that is believed to be secure for (only) one hour. Minimizing the length of the RSA modulus can result in a substantial performance improvement for RSA operations. This is indicated in Table 1, which shows RSA timings in milliseconds (ms) for a range of different modulus sizes. The tests described in Table 1 were performed on a SPARCstation 10 using a public exponent equal to three.

By using small short-term keys and changing them frequently, we have achieved reasonable performance for our reliable and atomic group multicast. Some per-

Table 1: CryptoLib RSA timings (ms), SPARC 10

	modulus length (bits)					
	300	400	500	600	700	800
sign	18.3	33.3	49.9	83.3	118.3	178.3
verify	1.3	1.8	2.2	2.5	3.3	4.0

formance measurements for reliable and atomic multicast are shown in Figures 2 and 3. The tests described in these figures were performed between user processes over SunOS 4.1.3 on moderately loaded SPARCstation 10s spanning several networks. In these tests, the RSA modulus of a short-term key was 300 bits in length.[4]

Figure 2 illustrates mean reliable multicast latency in milliseconds as a function of group size, for 0, 1, and 4 kilobyte (kb) messages. Despite the use of 300 bit RSA moduli, public key operations still dominate the latency of reliable multicast, at least for small messages. For example, reliably multicasting a message to a group of size four incurs the following public key operations on the critical path of the protocol: one digital signature (each member signing an echo message); two signature verifications by the multicast initiator (to verify two incoming echo messages); and three signature verifications at each other member (to verify the echo messages in the commit message). Reading from Table 1, these operations account for roughly 25 ms, or 60% of the latency reported for a 0 kb message in Figure 2. The remaining 17 ms is primarily communication costs. Obviously, the latency of the protocol could be greatly reduced by using special-purpose hardware to perform RSA operations. Such hardware could also make it feasible for the init message of the echo multicast protocol to be authenticated via digital signatures (versus over point-to-point authenticated channels), which opens the possibility of communicating this message by hardware multicast.

Mean latency of atomic multicast is not included in Figure 2 because this quantity typically would bear little relation to the actual latency experienced per A-multicast. In general, the latency for an atomic multicast can vary widely depending on the policy by which the sequencer reliably multicasts order messages. Due to the overhead imposed by a reliable multicast, this policy should minimize the number of order messages R-mcast, while still ensuring a reasonable latency for each atomic multicast. For instance, in our present implementation, once the sequencer R-mcasts an order message, it waits to R-mcast another order message until either it has R-delivered five new a-msg messages or a timeout has passed since it R-delivered a new a-msg

[4] A 300 bit RSA modulus should be secure for roughly an hour against an adversary with the computational resources used in the recent factorization of RSA-129 (A. Odlyzko, private communication, May 1994).

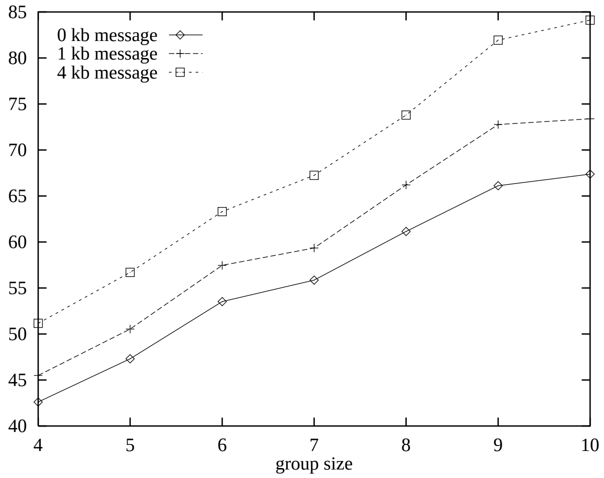

Figure 2: Reliable multicast latency (ms)

message, whichever comes first. Given this policy, the latency of *A-mcast*(m) can be as little as twice the latency of reliable multicast, if $\langle$a-msg:$m\rangle$ were the fifth such message R-delivered at the sequencer since the sequencer last R-mcast an order message. However, the latency could also be greater, e.g., if $\langle$a-msg:$m\rangle$ were the first message R-delivered at the sequencer after the sequencer R-mcast an order message. We are presently experimenting with a variety of policies for reliably multicasting order messages, to determine the best policies for various workloads.

Figure 3 shows the mean sustainable throughput for reliable and atomic multicast as a function of group size. Those curves marked "one multicasting" describe tests in which one group member repeatedly multicast (0 kb) messages, reliably or atomically depending on the curve. In the atomic multicast tests, the multicasting member was different from the sequencer. (The case in which the multicasting member is the sequencer performs roughly the same as one member R-mcasting, because the sequencer can piggyback all ordering information on its own a-msg messages; see Section 6.) The curves marked "all multicasting" describe tests in which all group members repeatedly multicast to the group. It is interesting to note that atomic multicast achieves almost the throughput of reliable multicast when all members are multicasting. This results from the sequencer piggybacking much of the ordering information on its own a-msg messages, thereby limiting the number of separate order messages that it R-mcasts.

9 Conclusion

We have presented new reliable and atomic group multicast protocols for asynchronous distributed systems that can suffer malicious process corruptions.

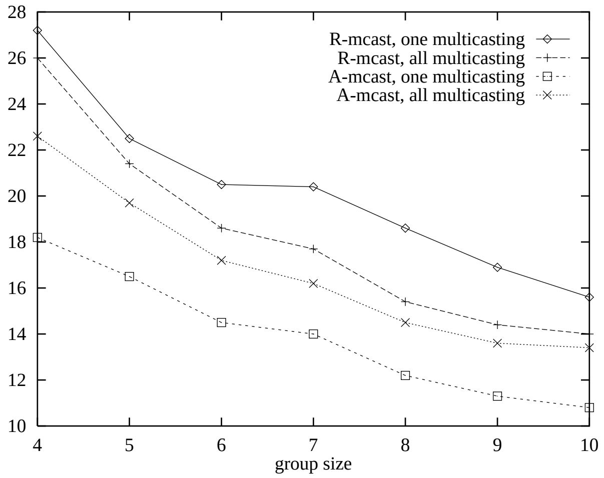

Figure 3: Throughput (deliveries/s)

These protocols are central to Rampart, a toolkit of protocols for the construction of distributed services that retain their integrity and availability despite the penetration of some component servers by an attacker. To our knowledge, Rampart is the first toolkit of its kind, and the first system to demonstrate the feasibility of reliable and atomic multicast in asynchronous systems subject to process corruptions. We continue to optimize our protocol implementations, although current performance numbers indicate that these protocols can provide performance that suffices for many types of applications. This is especially true if special-purpose hardware is available for performing cryptographic operations.

As discussed in Section 1, Rampart was motivated by a larger effort to provide support for comprehensive security mechanisms in loosely coupled, large-scale distributed systems. We are presently constructing security services with Rampart as part of this effort. We also anticipate that our reliable and atomic group multicast protocols may be of use in other types of secure distributed computing. We hope to report on our experiences with these efforts in future papers.

Acknowledgements

We thank Matt Blaze, Matt Franklin, Jack Lacy, Tom London, Stuart Stubblebine, and the anonymous referees for their comments on drafts of this paper.

References

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

[2] D. Beaver. Multiparty protocols tolerating half faulty processors. In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89 Proceedings* (Lecture Notes in Computer Science 435), pages 560–572. Springer-Verlag, 1990.

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

[4] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, April 1989.

[5] J. Chang and N. F. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[6] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 11–19, May 1988.

[7] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, pages 200–206, June 1985. A revised version appears as IBM Research Laboratory Technical Report RJ5244 (April 1989).

[8] 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, April 1985.

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

[10] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[11] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 218–229, May 1987.

[12] K. R. Iversen. A cryptographic scheme for computerized general elections. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91 Proceedings* (Lecture Notes in Computer Science 576), pages 405–419. Springer-Verlag, 1992.

[13] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[14] 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, October 1993.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[16] S. W. Luan and V. D. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):271–285, July 1990.

[17] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[18] J. H. Moore. Protocol failures in cryptosystems. *Proceedings of the IEEE*, 76(5), May 1988.

[19] F. M. Pittelli and H. Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems*, 7(1):25–60, February 1989.

[20] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 73–85, May 1989.

[21] M. K. Reiter. A secure group membership protocol. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 176–189, May 1994.

[22] M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[23] M. K. Reiter, K. P. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. To appear in *ACM Transactions on Computer Systems*, 1994.

[24] R. L. Rivest. The MD4 message digest algorithm. In A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology—CRYPTO '90 Proceedings* (Lecture Notes in Computer Science 537), pages 303–311. Springer-Verlag, 1991.

[25] R. L. Rivest. *RFC 1321: The MD5 Message Digest Algorithm.* Internet Activities Board, April 1992.

[26] 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, February 1978.

[27] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 561–568, May 1993.

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

[29] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, and A. Tully. Principal features of the VOLTAN family of reliable node architectures for distributed systems. *IEEE Transactions on Computers*, 41(5):542–549, May 1992.

[30] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, August 1984.

[31] 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*, April 1992.

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

A Proofs

In this appendix, we sketch the proofs of Agreement and Validity-1 for the reliable multicast protocol of Section 5. As described in Section 2, these proofs rely on the assumption that at least $\lceil (2|V^x|+1)/3 \rceil$ members of each V^x are honest.

Lemma 4 *If V^{x+1} is generated and an honest $p \in \bigcap_{k\geq 0} V^{x+k}$ never executes E-deliver$(q, x, \langle \mathsf{end} \rangle)$ for some $q \in V^x$, then there exists a $k > 0$ such that $q \notin V^{x+k}$.*

Proof. (Sketch.) First suppose that q is honest and, for the contrapositive, that $q \in V^{x+k}$ for all $k \geq 0$. Then, q eventually receives V^{x+1} and E-mcasts its end for view x. Since no honest process could have already R-delivered V^{x+k} for any $k > 0$ (having not received a flush for view $x+k$ from p or q), at least $\lceil (2|V^x|+1)/3 \rceil$ honest members of V^x send echo messages for this multicast and the echo multicast completes. So, p E-delivers an end for view x from q.

Now suppose that q is corrupt, and that p never executes *E-deliver*$(q,\ x,\ \langle \mathsf{end} \rangle)$. If some honest $r \in V^x$ E-delivers an end for view x from q, then r eventually forwards $\langle \mathsf{commit}{:}q, x, \langle \mathsf{end} \rangle \ldots \rangle$ to p (see Section 4.2). Since p never E-delivers an end from q, p must reject this commit because it received a view V^{x+k} such that $q \notin V^{x+k}$, and we have the result. Now suppose that no honest member of V^x E-delivers an end from q. Since p does not execute *E-mcast*$(x+1, \langle \mathsf{flush}{:}\ * \rangle)$ until it receives some V^{x+k} such that $q \notin V^{x+k}$, and since $p \in V^{x+k}$ for all $k \geq 0$, no honest process can R-deliver V^{x+k} or execute *adds*$(x+k)$ for any $k > 0$ until some V^{x+k} such that $q \notin V^{x+k}$ is generated. So, for all $k > 0$, if $q \in \bigcap_{0 \leq k' \leq k} V^{x+k'}$ then $V^{x+k+1} \subseteq V^{x+k}$. Moreover, for all $k > 0$ such that $q \in \bigcap_{0 \leq k' \leq k} V^{x+k'}$, all honest members of $V^x \cap V^{x+k}$ execute *remove*$(x+k, q)$. Since these honest members of $V^x \cap V^{x+k}$ account for at least $\lceil (2|V^{x+k}|+1)/3 \rceil - 1 \geq \lfloor (|V^{x+k}|-1)/3 \rfloor + 1$ honest members of V^{x+k}, the result follows. □

Lemma 5 *If an honest $p \in \bigcap_{k \geq 0} V^{x+k}$ never executes E-deliver$(q, x, \langle \mathsf{flush}{:}\ * \rangle)$ for some $q \in V^x$, then there exists a $k > 0$ such that $q \notin V^{x+k}$.*

Proof. (Sketch.) We prove the result by induction on x: suppose the result holds for all V^y, $y \leq x$, and consider view $x+1$. First suppose that $q \in V^{x+1}$ is honest and, for the contrapositive, that $q \in V^{x+k}$ for all $k > 0$. If $q \notin V^x$ (or if $x = 0$), then q E-mcasts its flush for view $x+1$ when it receives V^{x+1}. If $q \in V^x$, then since q receives V^{x+k} for all $k > 0$, the induction hypothesis and Lemma 4 imply that q E-mcasts its flush for view $x+1$. Since no honest process could have already R-delivered V^{x+k} for any $k > 0$, the echo multicast completes, and p E-delivers a flush from q for view $x+1$.

Now suppose that q is corrupt, and that p never E-delivers a flush for view $x+1$ from q. If an honest $r \in V^{x+1}$ E-delivers a flush for view $x+1$ from q, then r eventually forwards $\langle \mathsf{commit}{:}q, x+1, \langle \mathsf{flush}{:} \ldots \rangle \ldots \rangle$ to p (see Section 4.2). Since p never E-delivers a flush from q, p must reject this commit message because it received a view V^{x+k} such that $q \notin V^{x+k}$, and we have the result. So, now suppose that no honest member of V^{x+1} E-delivers a flush for view $x+1$ from q. Since no honest member of V^{x+1} can R-deliver V^{x+1} or thus execute *adds*$(x+k)$ for any $k > 0$ without first receiving some V^{x+k} such that $q \notin V^{x+k}$, it follows that for all $k > 0$, if $q \in \bigcap_{0 < k' \leq k} V^{x+k'}$ then

$V^{x+k+1} \subseteq V^{x+k}$. If $x = 0$, then each honest member of V^{x+1} E-mcasts its flush for view $x+1$ and executes *remove*$(x+1,\ q)$. Since for all $k > 0$ such that $q \in \bigcap_{0<k'\le k} V^{x+k'}$, these honest members account for $\lceil (2|V^{x+k}|+1)/3 \rceil \ge \lfloor (|V^{x+k}|-1)/3 \rfloor + 1$ members of V^{x+k}, the result follows. If $x > 0$, then by the induction hypothesis and Lemma 4, there exists a $\hat{k} > 0$ such that each honest member of $\bigcap_{0\le k'\le \hat{k}} V^{x+k'}$ E-mcasts its flush for view $x+1$ and thus executes *remove*$(x+1,\ q)$. Since $V^{x+k} = \bigcap_{0<k'\le k} V^{x+k'}$ for each $k \ge \hat{k}$ such that $q \in \bigcap_{0<k'\le k} V^{x+k'}$, these honest members account for at least $\lceil (2|V^{x+k}|+1)/3 \rceil - 1 \ge \lfloor (|V^{x+k}|-1)/3 \rfloor + 1$ members of V^{x+k}, and we have the result. □

Theorem 1 *(Validity-1) If p is an honest member of V^{x+k} for all $k \ge 0$, then p executes R-deliver(V^x).*

Proof. (Sketch.) The proof is by induction on views. For the induction step, suppose that the result holds for V^x, and consider V^{x+1}. Lemmas 4 and 5 imply that p E-mcasts a flush for view $x+k$ for all $k > 0$. Moreover, any $q \in V^{x+1}$ from which p does not E-deliver a flush message for any view $x+k$ is eventually removed (Lemma 5). Since V^{x+1} is finite, p must eventually R-deliver V^{x+1}. □

Lemma 6 *If an honest $p \in \bigcap_{k\ge 0} V^{x+k}$ executes R-deliver(r,m) in view x, then all honest $q \in V^x$ that execute R-deliver(V^{x+1}) also execute R-deliver(r,m) in view x.*

Proof. (Sketch.) Suppose that $p \in \bigcap_{k\ge 0} V^{x+k}$ and $q \in V^x$ are honest. For q to R-deliver V^{x+1}, there must be a $k_q > 0$ such that for every $r \in \bigcap_{0<k'\le k_q} V^{x+k'}$, q E-delivers a flush from r for each view $V^{x+1} \ldots V^{x+k_q}$. In particular, q must E-deliver a flush from p for each view $V^{x+1} \ldots V^{x+k_q}$ before R-delivering V^{x+1}.

We first show that if before R-delivering V^{x+1}, p executes *E-deliver*$(r,\ x+k,\ \langle \mathsf{flush}\colon S\rangle)$ where $r \in \bigcap_{0\le k'\le k} V^{x+k'}$ and $0 < k \le k_q$, and if this is p's first execution of *E-deliver*$(r,\ x+k,\ \langle \mathsf{flush}\colon *\rangle)$, then q also executes *E-deliver*$(r,\ x+k,\ \langle \mathsf{flush}\colon S\rangle)$ before R-delivering V^{x+1}. We prove this by induction on $k_q - k$. If $k = k_q$, then the result follows immediately because $r \in \bigcap_{0\le k'\le k_q} V^{x+k'} \subseteq \bigcap_{0<k'\le k_q} V^{x+k'}$. Now suppose that $k < k_q$, and that *E-deliver*$(r,\ x+k,\ \langle \mathsf{flush}\colon S\rangle)$ is the l-th E-delivery from r for view $x+k$ at p. It suffices to show that q executes at least l E-deliveries from r for view $x+k$ before R-delivering V^{x+1}. So, consider any $l' \le l$, and suppose that *E-deliver*$(r,\ x+k,\ m)$ for some m is the l'-th E-delivery from r for view $x+k$ at p. If p added $\langle \mathsf{commit}\colon r, x+k, m\ldots\rangle$ to *commits*$^{x+k}$ in step 4 of the the echo multicast protocol of Section 4.1, then p forwards this commit to q in its flush for view $x+k+1$ (unless the l'-th echo multicast from r for view $x+k$ is stable). So q adds this commit to *commits*$^{x+k}$ before R-delivering V^{x+1}. Now suppose that p added $\langle \mathsf{commit}\colon r, x+k, m\ldots\rangle$ to *commits*$^{x+k}$ in step 5a of the reliable multicast protocol of Section 5, i.e., due to executing *E-deliver*$(r',\ x+k+1,\ \langle \mathsf{flush}\colon S'\rangle)$ for some $r' \in \bigcap_{0\le k'\le k+1} V^{x+k'}$ and some S' such that $\langle \mathsf{commit}\colon r, x+k, m\ldots\rangle \in S'$. By the induction hypothesis, q executes *E-deliver*$(r',\ x+k+1,\ \langle \mathsf{flush}\colon S'\rangle)$ before R-delivering V^{x+1}, and thus adds $\langle \mathsf{commit}\colon r, x+k, m\ldots\rangle$ to *commits*$^{x+k}$ before R-delivering V^{x+1}.

To prove the lemma, now suppose that p executes *R-deliver*$(r,\ m)$ in view x, and that *E-deliver*$(r,\ x,\ \langle \mathsf{r\text{-}msg}\colon m\rangle)$ is the l-th E-delivery from r for view x at p. To show that q executes *R-deliver*(r,m) in view x, it suffices to show that q executes at least l E-deliveries from r for view x. So, consider any $l' \le l$, and suppose that *E-deliver*$(r,\ x,\ m')$ for some m' is the l'-th E-delivery from r for view x at p. If p added $\langle \mathsf{commit}\colon r, x, m'\ldots\rangle$ to *commits*^{x} in step 4 of the the echo multicast protocol of Section 4.1, then p forwards this commit to q in its flush for view $x+1$ (unless the l'-th echo multicast from r for view x is stable). Now suppose that p added $\langle \mathsf{commit}\colon r, x, m'\ldots\rangle$ to *commits*^{x} in step 5a of the reliable multicast protocol of Section 5, i.e., due to executing *E-deliver*$(r',\ x+1,\ \langle \mathsf{flush}\colon S\rangle)$ for some $r' \in V^x \cap V^{x+1}$ and some S such that $\langle \mathsf{commit}\colon r, x, m'\ldots\rangle \in S$. In this case, the previous paragraph shows that q also executes *E-deliver*$(r',\ x+1,\ \langle \mathsf{flush}\colon S\rangle)$ before R-delivering V^{x+1}, and thus adds $\langle \mathsf{commit}\colon r, x, m'\ldots\rangle$ to *commits*^{x}. □

Theorem 2 *(Agreement) If p and q are honest members of V^{x+k} for all $k \ge 0$ and p executes R-deliver(r,m) in view x, then q executes R-deliver(r,m) in view x.*

Proof. (Sketch.) By Theorem 1, q R-delivers V^x. If V^{x+1} is never generated, then the result follows because p forwards $\langle \mathsf{commit}\colon r, x, \langle \mathsf{r\text{-}msg}\colon m\rangle, \ldots\rangle$ to q if necessary (see Section 4.2). If V^{x+1} is generated, then q R-delivers V^{x+1} by Theorem 1 and thus R-delivers m in view x by Lemma 6. □