A High-Throughput Secure Reliable Multicast Protocol

Dahlia Malkhi Michael Reiter

AT&T Labs—Research, Murray Hill, New Jersey, USA
{dalia,reiter}@research.att.com

Abstract

A (secure) reliable multicast protocol enables a process to multicast a message to a group of processes in a way that ensures that all honest destination-group members receive the same message, even if some group members and the multicast initiator are maliciously faulty. Reliable multicast has been shown to be useful for building multiparty cryptographic protocols and secure distributed services. We present a high-throughput reliable multicast protocol that tolerates the malicious behavior of up to fewer than one-third of the group members. Our protocol achieves high-throughput using a novel technique for chaining *multicasts, whereby the cost of ensuring agreement on each multicast message is amortized over many multicasts. This is coupled with a novel flow-control mechanism that yields low multicast latency.*

1. Introduction

Reliable multicast is a fundamental communication protocol that underlies many forms of secure distributed computation. A (secure) reliable multicast protocol enables a process to multicast a message to a group of processes in a way that ensures that all honest destination-group members receive the same message, despite the contrary efforts of potentially malicious group members and even a malicious multicast initiator. Reliable multicast has been shown to be useful for constructing multiparty cryptographic protocols that enable systems to operate correctly despite the malicious (Byzantine) behavior of some components [7]. Practical examples of this can be found in Ω, a distributed, penetration-tolerant key management service that we are developing at AT&T [14]. Ω makes use of distributed computations to perform key backup, recovery, and other functions in a way that ensures the correctness and availability of these functions, while hiding sensitive information from any sufficiently small coalition of penetrated servers. Reliable multicast underpins the protocols used to communicate intermediate results reported by servers and helps to ensure that correct servers take consistent actions.

In this paper we present a new, practical reliable multicast protocol that is suitable for use in asynchronous distributed systems and that can tolerate the malicious behavior of up to fewer than one-third of the destination group members. The main contributions of this protocol are two mechanisms for maximizing multicast throughput (*i.e.*, deliveries per second) and for maintaining low multicast latency (*i.e.*, the time between multicast initiation and its delivery). These mechanisms significantly improve the message complexity of previously known techniques for reliable multicast (*e.g.*, [4, 12]) in the case of no failures, which should be the common case in most systems. Preliminary performance measurements conducted in our laboratory on a prototype implementation of this protocol show encouraging results: Our prototype sustains a steady throughput of 250 1-kilobyte messages per second among eight Sparcstation 20s, using hardware broadcast on a 10 Mbit/s Ethernet.

The novel mechanisms by which we achieve high performance in our protocol are based on two principles. The factor limiting performance in previous, practical reliable multicast protocols is the cost of computing digital signatures (*e.g.*, using RSA [16]) on message acknowledgements. Our first principle thus attempts to amortize the cost of computing a digital signature over many multicasts by a technique called *acknowledgement chaining*. Briefly, in this technique a single digital signature serves to acknowledge messages transitively, *i.e.*, an acknowledgement (signature) of one message M serves also to acknowledge the messages that M acknowledges, and so on. This method of chaining acknowledgements was influenced by prior work in benignly fault-tolerant systems, notably the Trans-Total [10] and Transis [5] systems. Chaining (or *linking*) was also used in [2] for timestamping documents by establishing their place among a sequence of similarly timestamped documents.

The second principle behind our protocol is intended to limit the latency of multicasts given this chaining technique. The latency of a multicast is determined primarily by the number of signatures on the critical path between initiation and delivery. If acknowledgement chaining is employed in an uncontrolled manner, it could result in all signatures required to deliver a multicast being performed serially, with severe consequences for the latency of that multicast. The second contribution of our protocol is a flow control mechanism that maximizes concurrency in the preparation of the acknowledgements required to deliver each multicast. In this way, the latency of each multicast is minimized without sacrificing the throughput gained with the acknowledgement chaining technique.

In this paper, we state our protocol assuming a static set of processes. It is possible, however, to use known techniques to extend our protocol to operate in a dynamic environment in which processes may leave or join the set of destination processes and in which processes may fail and recover. In particular, using techniques similar to those of [12], our protocol naturally extends to support a *virtually synchronous* communication environment, which has been shown to simplify the development of distributed programs [3]. These extensions also support more effective failure handling and garbage collection than we describe here.

The rest of this paper is structured as follows. In Section 2 we describe our assumptions about the system. In Section 3 we describe the semantics of our protocol. In Section 4 we detail the acknowledgement chaining technique, and in Section 5 we describe how to extend the resulting protocol with a flow-control mechanism for maximizing the performance of our protocol. We conclude in Section 6.

2. System model

We assume a system consisting of a static set of n *processes* $p_0, p_1, \ldots p_{n-1}$. We will often use p, q, r, and s to denote processes when subscripts are unnecessary. A process that behaves according to its specification is *honest*. A *corrupt* process, however, may behave in any fashion whatsoever (Byzantine failures), constrained only by the assumptions stated below. Corrupt processes include those that fail benignly. Our protocol requires that at most $\lfloor (n-1)/3 \rfloor$ processes are corrupt, and thus that at least $\lceil (2n+1)/3 \rceil$ processes are honest.

Processes communicate exclusively via a completely connected, point-to-point network. Communication channels between honest processes are FIFO and reliable, in the sense that if the sender and destination of a message are honest, then the destination eventually receives the message. However, communication is *asynchronous*, in the sense that there is no known finite bound on message transmission times. The communication channel between each pair of processes is authenticated and protects the integrity of communication (*e.g.*, using well-known cryptographic techniques [17]), so that a receiver can tell the channel on which a message is received.

Each process p possesses a private key known only to itself, with which it can digitally sign sets of messages (*e.g.*, [16]). A set B signed by p is denoted $\{B\}_p$. We often use $\{M\}_p$ as an abbreviation for $\{B\}_p$ where $M \in B$, *i.e.*, for the signature of a set containing M. We assume that each process can obtain the public keys of other processes as needed, with which it can verify the origin of signed sets of messages. Henceforth, we mention explicitly when message signing is used; otherwise, messages (or portions thereof) are sent unsigned.

3. Protocol semantics

In this section we more carefully state the semantics of our reliable multicast protocol. Our 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). It is convenient to assume that an honest process does not R-mcast the same message twice; this can be enforced, *e.g.*, by the process including a sequence number in each message. As described in the Introduction, the task of a reliable multicast protocol is to ensure that processes deliver the same messages. More precisely, our protocol satisfies the following properties.

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

Agreement: If p and q are honest and p executes *R-deliver*(r, m), then q executes *R-deliver*(r, m).

Validity: If p and q are honest and p executes *R-mcast*(m), then q executes *R-deliver*(p, m).

Source Order: If p and q are honest and both execute *R-deliver*(r, m) and *R-deliver*(r, m'), then they do so in the same relative order.

Note that *Agreement* and *Source Order* together imply that for any l and any process r, the l'th R-delivery from r is the same at all honest processes. In addition, while here we present our protocol in a way that allows multicasts only from the processes $p_0, \ldots, p_{n-1}$, it is possible to extend the protocol to allow multicasts from outside the destination group (*e.g.*, in the manner of [13]).

The above semantics distinguish the problem we are attempting to solve from other problems studied in the scientific literature on secure and fault-tolerant distributed computing. In particular, our specification is weaker than the well-studied problem of Byzantine agreement [9]. The Byzantine agreement problem is as follows: The honest members of a system must irreversibly decide on a value sent by a designated sender s among them, such that (i) every honest member decides on exactly one value, (ii) no two honest members decide on different values, and (iii) if the sender s is honest, then each honest member decides on the value sent by s. It is tempting to think of reliable multicast as multiple instances of Byzantine agreement with varying senders, where the "decision" is the R-delivery of a sender's message. However, Byzantine agreement is strictly stronger, in that it requires a decision to be reached at honest members even in the case of a faulty sender ((i) above). In contrast, reliable multicast does not require honest processes to R-deliver messages from a faulty process. Reliable multicast is also weaker than atomic (totally-ordered) multicast (e.g., [11, 12, 13]). This problem imposes an ordering requirement that is stronger than Source Order, *i.e.*, that honest processes execute the same totally-ordered sequence of multicast deliveries. Due to their stronger properties, neither Byzantine agreement nor atomic multicast is solvable in asynchronous systems (implied by [6]), whereas reliable multicast is.

4. Chain multicast

In this section, we describe a subprotocol that will be used in our reliable multicast protocol. This protocol, called *chain multicast*, provides an interface that enables a process to multicast a message to the processes. Chain multicast can itself be used as a reliable multicast protocol, i.e., it satisfies the Integrity, Agreement, Validity and Source Order properties described in Section 3. We nevertheless build another reliable multicast protocol over it in Section 5 that offers better performance when no failures occur. The chain multicast protocol takes its name from the technique of *acknowledgement chaining* that was developed in prior work on benignly fault-tolerant systems such as Trans-Total [10] and Transis [5]. We outline this technique in Section 4.1 and describe the protocol in Section 4.2.

4.1. Acknowledgement chaining

The principle of acknowledgement chaining works by processes sending messages to the group of processes. Each message M is a tuple of the form

$$M = \langle p, m, \{B\}_p \rangle \tag{1}$$

where p is a process identifier, m is the (application-specific) contents of the message, and B is a set to be described below. If the sending process is honest, then p is the identifier of that sending process and included in m is a non-negative integer header, denoted $seq(m)$, that denotes the sequence number of the message m.

The set B contains *message digests*. A message digest function D maps any arbitrary length input x to a fixed length output $D(x)$ and has the property that it is computationally infeasible to determine two inputs x and x' such that $D(x) = D(x')$. Thus, for all practical purposes, the digest $D(x)$ uniquely identifies x, and we assume this for the remainder of the paper. Several efficient message digest functions have been proposed (*e.g.*, MD5 [15]).

B can be viewed as acknowledgements of other messages, *i.e.*, if $D(M') \in B$ (and p is honest) then p has received M', and we say that p (directly) acknowledges M'. The acknowledgements in messages naturally induce a relation, which we denote $\rightarrow$, among message digests. That is, given M of the form (1) and M', $D(M) \rightarrow D(M')$ if and only if $D(M') \in B$. An *acknowledgement chain* for M' from p, denoted $chain_p(M')$, is a sequence of messages $M_0, \ldots, M_k$, $k \geq 0$, and a signature $\{D(M_0)\}_p$, such that $M_k = M'$, and $D(M_i) \rightarrow D(M_{i+1})$ for all $0 \leq i < k$. Intuitively, if there is an acknowledgement chain for M' from p, then p has received M'.

Given this machinery, the protocol executes roughly as follows. For a process p to send a message m, it sends (1), where B contains digests of some messages that it has received. The protocol derives its efficiency by including only those digests that are necessary to acknowledge, either directly or by chaining, every message that p has received (and not already acknowledged). To implement this, as a process receives messages, it builds a directed graph whose nodes are message digests and whose edges are the relation $\rightarrow$. That is, when a process receives (1), it inserts the node $D(M)$ and, for each $d \in B$, the node d and the edge $D(M) \rightarrow d$ into its graph. When a process sends a message of the form (1), it includes in B the digests (nodes) of unacknowledged messages, i.e., the nodes appearing in the graph without any nodes pointing at them. To deliver a message m to the application, p waits until there are acknowledgement chains for this message in its graph from $\lceil (2n+1)/3 \rceil$ processes. The number $\lceil (2n+1)/3 \rceil$ is significant because if there are acknowledgement chains for a message from $\lceil (2n+1)/3 \rceil$ processes and there are at most $\lfloor (n-1)/3 \rfloor$ corrupt processes, then there are acknowledgement chains for this message from a majority of the *honest* processes. Provided that an honest process forms an acknowledgement chain to at most one message with the same process identifier and sequence number, no two honest processes can deliver messages with the same sender and sequence number but with different contents.

The acknowledgement-chaining principle allows each message to be directly acknowledged only a few times, and through chains of acknowledgements, to be indirectly acknowledged by other processes. This leads to an efficient utilization of resources. For example, four processes p, q, r, s could communicate as follows:

$$\begin{aligned}
M_1 &= \langle p, m_1, \{\} \rangle \\
M_2 &= \langle q, m_2, \{D(M_1)\}_q \rangle \\
M_3 &= \langle r, m_3, \{D(M_2)\}_r \rangle \\
M_4 &= \langle p, m_4, \{D(M_3)\}_p \rangle \\
M_5 &= \langle s, m_5, \{D(M_4)\}_s \rangle
\end{aligned}$$

In this scenario, messages M_1, M_2 are deliverable, since there are acknowledgement chains for each from three processes. But only one explicit signed acknowledgement was sent for any message.

This technique guarantees the *uniqueness* of messages, *i.e.*, that when two honest processes deliver the l'th message from some process q, they in fact deliver the same message. Moreover, if up to $\lfloor (n-1)/3 \rfloor$ processes fail benignly, messages will continue to be delivered. However, even a single corrupt process can prevent progress in this protocol. For example, if p is corrupt above, it could send conflicting "versions" of M_1 (i.e., messages with the same sender and sequence number but different contents) to q, r, and s, say M_1' to q and M_1'' to r and s. Once q sends M_2 acknowledging M_1', the graphs shown in Figure 1 result, where $d' = D(M_1')$ is, to r and s, a digest of an unknown message. Since M_2 acknowledges a message that r and s did not receive, r and s cannot acknowledge M_2, and thus M_2 will never be delivered. To make progress, the protocol below takes steps to detect corrupt members and bypass undeliverable messages.

4.2. The protocol

In this section, we detail the chain multicast protocol. This protocol provides an interface *C-mcast*(m) by which a process p multicasts a message m. In addition, it provides a *C-sign*() interface

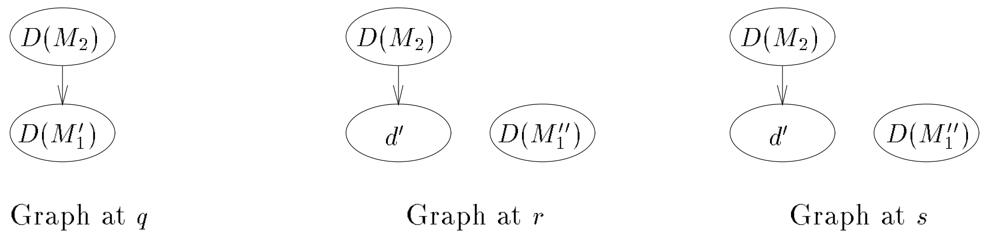

Figure 1. Conflicting messages M_1', M_1'' prevent M_2 from being delivered

routine that prepares a signature for digests of messages received by p. Under normal conditions, a user should invoke *C-sign* immediately before *C-mcast*. The reasons for separating the two operations will become clear in Section 5. A process delivers a message m from q via the chain multicast protocol by executing *C-deliver*(q, m).

The protocol is implemented using messages of the form (1). Given such a message, it is convenient to define

$$\begin{aligned} \mathit{sender}(M) &= p \\ \mathit{seq}(M) &= \mathit{seq}(m) \\ \mathit{payload}(M) &= m \end{aligned}$$

Two messages M and M' are *conflicting* if $\mathit{sender}(M) = \mathit{sender}(M')$, $\mathit{seq}(M) = \mathit{seq}(M')$, and $\mathit{payload}(M) \neq \mathit{payload}(M')$. Since some processes might be corrupt, it is possible throughout the course of our protocol that an honest process will receive conflicting messages. We say that a process C-delivers (or just delivers) M of the form (1) if it executes *C-deliver*(p, m).

As described in Section 4.1, each process maintains a graph, the nodes of which are message digests. A *shadow* message M at p is any undelivered message whose digest appears in p's graph but that p has not received on the channel from $\mathit{sender}(M)$. Shadow messages include messages not received at all, but whose digests appear in the graph because they were included in the acknowledgements of a received message. A *direct* message M at p is any undelivered message that is not a shadow, *i.e.*, that p has received from $\mathit{sender}(M)$. A *candidate* message at p is a message M such that all M', $D(M) \rightarrow D(M')$, are delivered. A *delivering set* for a message M is a set $\{\mathit{chain}_p(M)\}_{p \in P}$ where $|P| = \lceil (2n+1)/3 \rceil$. In addition to the graph, each process has a set variable B, initially empty, a value κ that is initially $\perp$ (null), and a vector of counters $\{c_i\}_{0 \leq i < n}$, indicating the sequence number of the last message it C-delivered from each process.

In stating our protocol, it is convenient to let $\rightarrow^*$ be the smallest relation satisfying (i) $d \rightarrow^* d$ for all d, and (ii) if $d_1 \rightarrow d_2$ and $d_2 \rightarrow^* d_3$, then $d_1 \rightarrow^* d_3$. The protocol at p executes as follows:

C–1 If *C-sign*() is executed, prepare a signature for B, i.e., set $\kappa = \{B\}_p$.

C–2 If *C-mcast*(m) is executed: Send

$$\langle p, m, \kappa \rangle$$

to each process. (This message is also received immediately at p and is treated according to the next rule.) Set κ to $\perp$.

C–3 If a message $M = \langle q, m, \{B_q\}_q \rangle$ is received from r:

1. If M is direct (*i.e.*, $r = q$) and there is a direct or delivered message that conflicts with M, then ignore and discard M and halt this routine.
2. Insert $D(M)$ and any $d \in B_q$ into the graph if they do not already appear.
3. Insert the edges $D(M) \rightarrow d$ into the graph for each $d \in B_q$.
4. If M is direct, then set B to contain exactly those nodes $D(\hat{M})$ in the graph such that (i) $\hat{M}$ is direct, (ii) for every $D(M')$ in the graph, if $D(\hat{M}) \rightarrow^* D(M')$, then M' is either already delivered or is a direct message in the graph, and (iii) there is no $D(M'')$ in the graph satisfying (i) and (ii) such that $D(M'') \rightarrow D(M)$.

C–4 Let M be a candidate message, where $sender(M) = p_j$. If a delivering set is received for M and $seq(M) = c_j + 1$:

1. Execute $\mathit{C\text{-}deliver}(p_j, payload(M))$, and set $c_j \leftarrow c_j + 1$.
2. After a timeout period has passed, send M and its delivering set to each process r that has not acknowledged delivering M.

To implement the acknowledgements required for C–4.2, each process piggybacks its vector $\{c_i\}_{0 \le i < n}$ on (*e.g.*, the payloads of) its chain multicasts.

4.3. Ensuring message C-delivery

The protocol described above ensures that honest processes will agree on the contents of C-delivered messages. Additional steps are required in order to guarantee that a message that is C-mcast will eventually be C-delivered. Some of the steps are obvious; *e.g.*, we have to specify that C-sign is called and C-mcasts are initiated periodically, in order to believe that acknowledgement chains from $\lceil (2n+1)/3 \rceil$ processes will be collected for any message. In addition, however, there are a number of subtle behaviors that can prevent C-mcasts from honest processes from ever being C-delivered.

1. As discussed before (Figure 1), it is possible that a C-mcast from an honest process might be prevented from becoming a candidate because it acknowledges a message from a corrupt process that can never be delivered. To ensure that a C-mcast from an honest process, say p, becomes a candidate, we use a simple retransmission scheme. That is, if after p executes *C-sign* and $\mathit{C\text{-}mcast}(m)$, this message does not become a candidate locally within some timeout period, then p executes $\mathit{C\text{-}mcast}(m)$ (without first executing $\mathit{C\text{-}sign}()$). This message is immediately a candidate at all honest processes. Although it is possible that both these transmissions of m can eventually become deliverable, the delivery counters will suppress duplicate delivery of m to the application at any honest member.

2. The prior rule ensures that a message from an honest process eventually becomes a candidate. There is still a risk, however, that a candidate message in an honest process' graph will not be C-delivered anywhere, even if it is from an honest process p, because no honest process receives a delivery set for it. This could happen, for example, if a corrupt process q sends conflicting

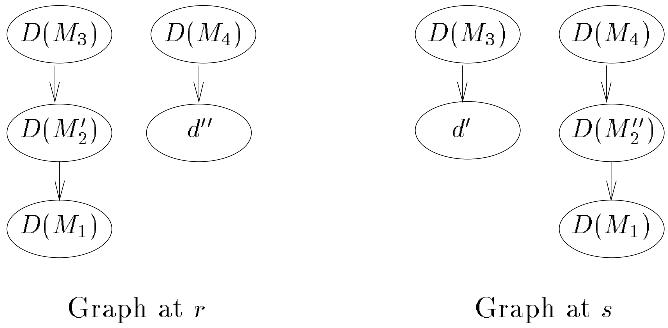

Figure 2. Conflicting messages M'_2, M''_2 prevent M_1 from being delivered

"versions" of a message M_2, each version acknowledging p's message M_1, to different honest members. Let M'_2 and M''_2 be these conflicting messages. Then, each honest process may indirectly acknowledge M_1 by directly acknowledging either M'_2 or M''_2. This is shown in Figure 2, where r has sent a message M_3 acknowledging M'_2, s has sent a message M_4 acknowledging M''_2, and $d'' = D(M''_2)$ and $d' = D(M'_2)$ are digests of unknown messages to r and s, respectively. As shown here, there may be only one acknowledgement chain for M_1 at each honest process.

In order to C-deliver a candidate message from an honest process in such cases, each honest process p must *directly* acknowledge each direct, candidate message if it is not delivered within some timeout period (even though p may have indirectly acknowledged it before).

More precisely, the code for chain multicast at p contains the following additional rules:

C–5 If *C-sign* or *C-mcast* is not executed within some timeout period, then execute *C-sign*, *C-mcast*$(\perp)$, where $\perp$ is an empty message (with a sequence number).

C–6 If *C-mcast*(m) is executed and $\langle p, m, \ldots\rangle$ does not become a candidate within some timeout period, then execute *C-mcast*(m) without first executing *C-sign*$()$.

C–7 If M is a direct candidate message, *C-deliver*$(sender(M), payload(M))$ is not executed within some timeout period, and p has not previously sent a message M' such that $D(M') \rightarrow D(M)$, then set $B = B \cup \{D(M)\}$ and immediately execute *C-sign*, *C-mcast*$(\perp)$.

For garbage collection purposes, our chain multicast protocol makes use of the concept of message *stability*; we say that a message is *stable* if it has been C-delivered at all honest processes. A process can determine that the l-th message from a process p_i is stable when it has delivered a vector from each process whose value for c_i is at least l. A message M (and the node $D(M)$ and any incident edges) where $sender(M) = q$ can be discarded when (i) some message M' such that $sender(M') = q$ and $seq(M') = seq(M)$ is stable, and (ii) every M'' such that $D(M) \rightarrow D(M'')$ is either stable or has been discarded already, meaning that M is no longer needed for any acknowledgement chain of another unstable message.

A limitation of this mechanism is that a corrupt process could prevent the removal of messages from the graphs of other processes, e.g., simply by failing to send stability updates. For more robust treatment of garbage collection, our protocol can be extended by known techniques for removing failed processes from the configuration (e.g., [12]). These extensions, however, are beyond the scope of this paper.

4.4 Proof of correctness

We now prove that the chain multicast protocol above satisfies Integrity, Agreement, Validity and Source Order (with *R-mcast* and *R-deliver* replaced with *C-mcast* and *C-deliver*, respectively). Thus, chain multicast is itself a reliable multicast protocol (though the reliable multicast protocol we build over it in Section 5 is more efficient).

Lemma 1 *If p is honest and $chain_p(M)$ exists, then M is either direct or delivered at p.*

Proof : Process p forms an acknowledgement chain for M only by inserting a digest d into B such that $d \rightarrow^* D(M)$, which can occur only in steps C–3.4 and C–7 of the protocol. A requirement for inserting d into B in step C–3.4 is that if $d \rightarrow^* D(M')$ then M' is direct or delivered. Similarly, step C–7 provides for only digests d of direct candidate messages—which by definition have the property that if $d \rightarrow^* D(M')$ then M' is direct or delivered—to be inserted into B. □

Lemma 2 (Integrity) *An honest process executes C-deliver(p, m) at most once and, if p is honest, only if p executed C-mcast(m).*

Proof : For an honest process q to execute *C-deliver*(p, m), q must obtain acknowledgement chains for $M = \langle p, m, \ldots \rangle$ from $\lceil (2n + 1)/3 \rceil$ processes, and in particular from at least one honest process. By Lemma 1, the first honest process to form an acknowledgement chain for M must have received M from p. If p is honest, this implies p executed *C-mcast*(m).

Duplicate deliveries are suppressed by delivering messages in the order of their sequence numbers. □

Lemma 3 *If an honest p executes C-deliver(r, m) and an honest q executes C-deliver(r, m') where $seq(m) = seq(m')$, then $m = m'$.*

Proof : For p to execute *C-deliver*(r, m), it must have received acknowledgement chains for $M = \langle r, m, \ldots \rangle$ from $\lceil (2n + 1)/3 \rceil$ processes, and similarly q must have received $\lceil (2n + 1)/3 \rceil$ acknowledgement chains for $M' = \langle r, m', \ldots \rangle$. Suppose for a contradiction that M and M' conflict. Prior to either of M or M' being delivered at any honest process, each honest process formed an acknowledgement chain to at most one of M or M', namely that which it first received directly from r (see Lemma 1). Therefore, acknowledgement chains from a majority of the honest processes, and thus from $\lceil (2n + 1)/3 \rceil$ processes, could exist for at most one of M and M'. □

Lemma 4 (Agreement) *If p and q are honest and p executes C-deliver(r, m), then q executes C-deliver(r, m).*

Proof : We prove the result by induction on the graph. By rule C–4.2, after p executes *C-deliver*(r, m), it either learns that q has executed *C-deliver*(r, m) as well (in which case we are done), or p sends $M = \langle r, m, \ldots\rangle$ and the acknowledgement chains that comprise its delivering set to q. By rules C–3.1, C–3.2, and C–3.3, messages included in these acknowledgement chains are inserted into q's graph. Moreover, by the induction hypothesis, eventually every M', $D(M) \rightarrow D(M')$, will be delivered by q, making M a candidate at q, and eventually $seq(M) = c_r + 1$ at q. Finally, note that by Lemma 3, q could not deliver some other m' with the same sequence number as m, and therefore, q will execute *C-deliver*(r, m). □

Lemma 5 (Validity) *If p and q are honest and p executes C-mcast(m), then q executes C-deliver(p, m).*

Proof : Rule C–6 guarantees that there is a message $M = \langle p, m, \ldots\rangle$ that becomes a candidate at some (and, by Agreement, all) honest processes. Also by Agreement, if any honest process delivers M, then all other honest processes do, and so it suffices to argue that some honest process delivers M. If a sufficiently long time passes without M being delivered, though, then all honest processes send direct acknowledgements for M (by C–7), thus forming at least $\lceil(2n + 1)/3\rceil$ acknowledgement chains to M. An honest process then delivers M. □

Lemma 6 (Source Order) *If p and q are honest and both execute C-deliver(r, m) and C-deliver(r, m'), then they do so in the same relative order.*

Proof : Both p and q deliver messages in the order of the sequence numbers in m and m'. □

5. The full reliable multicast protocol

In this section we describe the full reliable multicast protocol, which uses chain multicast from the previous section to deliver messages reliably and with high throughput. In the chain protocol, the promise of high throughput comes from the fact that a single digital signature can be used to acknowledge multiple messages (all of those included in the signed set, and acknowledged by those in the signed set), and thus that the cost of a digital signature can be amortized over many chain multicasts.

The latency of message delivery, however, may suffer significantly in this protocol, because to maximally amortize signing operations, the signature operations that are needed to deliver a message must be sequentialized. In today's computing environments, signature generation is a costly operation that is typically an order of magnitude slower than authenticated message transmission for most reasonable message sizes. For instance, the generation of an RSA [16] signature on a 75 MHz Sparcstation 20 using the CryptoLib software package [8] ranges from roughly 12 milliseconds (ms) for a (insecure) 300-bit RSA modulus,[1] to roughly 33ms for a (somewhat more secure) 512-bit modulus. These numbers are largely independent of the size of the message being signed, but still compare poorly to the roughly 1.5ms required for a 2 kilobyte message transmission over a 10Mbit/s Ethernet authenticated using (very secure) message authentication codes on such a

[1] A 300-bit RSA modulus should be secure for roughly an hour against an adversary with the computational resources used in the factorization of RSA-129 [1] (A. Odlyzko, private communication, May 1994). A 300-bit modulus should therefore be used in our protocol only if it is changed frequently, as in [13].

platform. Signature verification for RSA is faster: *e.g.*, with a public exponent of 3, verification takes 1ms for a 300-bit modulus and 1.5ms for a 512-bit modulus.

More generally, let S be the time it takes to sign a message, U be the time to verify a signature, and T be the typical time it takes to transmit a message of some constant size. The *delivery latency* of the chain multicast protocol is the time between a process p executing *C-mcast*(m) and some process executing *C-deliver*(p, m). In the most disadvantageous (but faultless) run of the chain multicast protocol, the delivery latency is $T + \lceil(2n+1)/3\rceil * (S+U+T)$. This occurs when signed acknowledgements are formed sequentially in a chain of $\lceil(2n+1)/3\rceil$ processes, each signature starting after the previous message in the chain is received. In this case, in a network of 9 Sparc 20s using 300-bit RSA moduli, the delivery latency would be roughly 105ms.

The latency of the chain protocol can be improved only by parallelizing the generation of signatures. The hope is to generate signatures in parallel at multiple processes, without sacrificing the chaining principle entirely. To achieve parallelization, the order of signing and transmission is regulated in the full reliable multicast protocol. The reliable multicast protocol operates by holding the transmission of messages that are *R-mcast*, and sending them via *C-mcast* according to the flow control policy. In addition, it is responsible for preparing signed acknowledgements of messages, such that signing is coordinated between different processes for efficiency, and such that signing delays message transmission as little as possible. In this manner, reliable multicast obtains its reliability from raw chain multicast and adds flow-control logic for better performance.

The reliable multicast protocol orders message transmission in a round-robin manner. Initially, p_0 is enabled to transmit a message. When a message is received from p_j, process $p_{(j+1) \bmod n}$ is enabled for transmission, and so on. Given this ordered transmission, $\frac{S}{T}$ consecutive transmissions can take place while message signing is performed elsewhere. Therefore, the protocol uses the following rule for signing digest acknowledgements. A process p_i that receives a message from sender $p_{(i-S/T) \bmod n}$—*i.e.*, from a sender that is $\frac{S}{T}$ hops preceding it in the order of transmission—begins preparing a signed acknowledgement for the messages currently in its graph (*i.e.*, for its set B). When process p_i becomes enabled, it C-mcasts a message with the already prepared signed acknowledgements ($\{B\}_p$). More precisely, the reliable multicast protocol at process p_i executes according to the following rules, where initially $sender = 0$ at p_i.

1. If *R-mcast*(m) is executed, put m in a *pending* queue.

2. Suppose that $sender = j$. If a message $M = \langle p_j, \ldots\rangle$ is received from p_j, or if a timeout period passes without such a message being received, then set $sender = (j+1) \bmod n$.

3. When $(i - sender) \bmod n$ becomes equal to $\frac{S}{T}$, execute *C-sign*().

4. When $sender = i$, dequeue the first message m in *pending*, and execute *C-mcast*(m).

5. If *C-deliver*(q, m) is executed, then execute *R-deliver*(q, m).

It is not difficult to verify that the properties of the chain multicast protocol proved in Section 4.4 extend also to this protocol, and so this protocol satisfies Integrity, Agreement, Validity, and Source Order.

A small example of how this protocol executes with five processes $p_0, \ldots, p_4$ and $\frac{S}{T} = 2$ is as follows:

$$
\begin{aligned}
M_0 &= \langle p_0, m_0, \{\} \rangle \\
M_1 &= \langle p_1, m_1, \{\} \rangle \\
M_2 &= \langle p_2, m_2, \{\} \rangle \\
M_3 &= \langle p_3, m_3, \{D(M_0)\}_{p_3} \rangle \\
M_4 &= \langle p_4, m_4, \{D(M_0), D(M_1)\}_{p_4} \rangle \\
M_5 &= \langle p_0, m_5, \{D(M_0), D(M_1), D(M_2)\}_{p_0} \rangle \\
M_6 &= \langle p_1, m_6, \{D(M_1), D(M_2), D(M_3)\}_{p_1} \rangle \\
M_7 &= \langle p_2, m_7, \{D(M_2), D(M_3), D(M_4)\}_{p_2} \rangle \\
&\vdots
\end{aligned} \tag{2}
$$

Note that each digest appears in $\frac{S}{T} + 1$ signed acknowledgements, and that each message contains only one signature (on possibly multiple digests).

Generally, under normal (faultless) conditions, this reliable multicast protocol can potentially achieve a delivery latency of $T + S + \lceil (2n+1)/3 \rceil * (U + T)$. This latency is derived as follows: The transmission of a message M takes T time. S is the time it takes for the process $\frac{S}{T}$ hops away from the sender to complete a signed acknowledgement for M, or equivalently for this process to become enabled to transmit. This is followed by $\lceil (2n+1)/3 \rceil$ transmissions, each one being initiated as soon as the previous one is received and thus taking T time, and each one containing a signed acknowledgement for M (direct or indirect). Finally, the $\lceil (2n+1)/3 \rceil$ signatures must be verified. This calculation assumes that each process is ready to transmit a message as soon as its turn arrives. In order for this to hold, it is assumed that $n \geq \frac{S}{T}$. In the example above, M_0 can be delivered after (i) it is transmitted, (ii) $\frac{S}{T} = 2$ transmissions (of M_1, M_2) occur, allowing signature preparation by p_3 to complete, and (iii) $\lceil (2n+1)/3 \rceil = 4$ more messages (M_3 through M_6) are transmitted and verified, with a total elapsed latency of $7T + 4U$.

A similar calculation with the more realistic performance numbers given above (with 300-bit RSA moduli and public exponents equal to 3) yields a delivery latency for 9 Sparc 20s of potentially between 30ms and 40ms.

6. Conclusion

In this paper we presented a high-throughput multicast protocol that ensures that all members of a multicast destination group receive the same multicast messages, despite the malicious collaboration of fewer than one-third of the group members. High throughput is achieved due to an acknowledgement chaining technique, whereby a single signature is used to indirectly acknowledge multiple messages. Our protocol also includes a flow control mechanism that enables concurrency in signing, in order to minimize multicast latency.

The performance of a communication protocol in an environment that admits intruders can be predicted only in the case of benign failures. Unfortunately, malicious processes may significantly slow down the system by inducing interruptions to the protocol. Future research will focus on means for detecting and preventing such attacks.

References

[1] D. Atkins, M. Graff, A. K. Lenstra, and P. C. Leyland. The magic words are squeamish ossifrage. In *Proceedings of Asiacrypt '94*, pages 219–229, 1994.

[2] D. Bayer, S. Haber and W.S. Stornetta. Improving the efficiency and reliability of digital time-stamping. *Journal of Cryptology* 3(2):99–111, 1991.

[3] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM* 36(12):37–53, December 1993.

[4] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM* 32(4):824–840, October 1985.

[5] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM* 39(4):64–70, April 1996.

[6] 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.

[7] 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.

[8] 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.

[9] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):328–401, July 1982.

[10] 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.

[11] L. E. Moser and P. M. Melliar-Smith. Total ordering algorithms for asynchronous Byzantine systems. In *Proceedings of the 9th International Workshop on Distributed Algorithms* (Lecture Notes in Computer Science 972), Springer-Verlag, September 1995.

[12] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

[13] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), pages 99-110, Springer-Verlag, 1995.

[14] M. K. Reiter, M. K. Franklin, J. B. Lacy, and R. N. Wright. The Ω key management service. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 38–47, March 1996. Revised version to appear in *Journal of Computer Security*.

[15] R. L. Rivest. RFC 1321: The MD5 message digest algorithm. Internet Activities Board, April 1992.

[16] 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.

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