The Rampart Toolkit for Building High-Integrity Services

Michael K. Reiter

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

Abstract. Rampart is a toolkit of protocols to facilitate the development of *high-integrity* services, i.e., distributed services that retain their availability and correctness despite the malicious penetration of some component servers by an attacker. At the core of Rampart are new protocols that solve several basic problems in distributed computing, including asynchronous group membership, reliable multicast (Byzantine agreement), and atomic multicast. Using these protocols, Rampart supports the development of high-integrity services via the technique of *state machine replication*, and also extends this technique with a new approach to server output voting. In this paper we give a brief overview of Rampart, focusing primarily on its protocol architecture. We also sketch its performance in our prototype implementation and ongoing work.

1 Introduction

Many techniques for enforcing security policy in distributed systems rely on trusted services for performing security-critical functions. Examples include authentication services or certification authorities to support cryptographic key distribution (e.g., [28, 18]) and access control services for the management and enforcement of access control policy (e.g., [13]). For these types of services to be trustworthy, they must be protected from tampering by attackers. In addition, since security and liveness in a surrounding system may rely on the availability of these services, they may need to be replicated for high availability. It has been argued that these requirements pose a conflict, in that replicating data or services for high availability makes them more difficult to protect [29, 15, 14, 18]. This tradeoff is particularly pertinent to security-critical services of a trusted computing base, as prudence dictates that these services be localized to facilitate their protection.

A compromise to balance the needs for security and availability in some services is to replicate the service in a way that enables it to remain correct and available despite the malicious penetration of some of its component servers by an attacker [15, 14, 24]. Techniques for building such *high-integrity* services have been explored primarily under the rubric

In *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), pp. 99–110, Springer-Verlag, 1995.

of Byzantine fault-tolerance. The most widely applicable technique for building high-integrity services is known as *state machine replication* [26]. In this approach, a service is implemented with multiple identical, deterministic servers, each initialized to the same state. Clients issue requests to the servers using an *atomic multicast* protocol, which ensures that all correct servers process the same requests in the same order and thus produce the same output for each request. If the correct servers sufficiently outnumber the faulty ones, then a client can identify the correct output by *output voting*, e.g., accepting the output returned by a majority of the servers.

A substantial impediment to using state machine replication in hostile environments is its reliance on atomic multicast. While many systems have demonstrated atomic multicast tolerant of *benign* process failures in general-purpose distributed systems (e.g., [20, 19, 3, 16, 1]), prior efforts have demonstrated atomic multicast tolerant of *malicious* failures only under the assumption that the underlying network is *synchronous*, i.e., that there are known bounds on message transmission times, processes' rates of execution, and relative clock drifts [7, 21, 27]. These synchrony assumptions make these solutions inappropriate for use in loosely-coupled systems, and especially for hostile environments in which messages can be delayed due to denial-of-service attacks [31].

We have undertaken the design and implementation of a protocol toolkit, called Rampart, to facilitate the construction of high-integrity services via state machine replication. A main focus of this effort has been to demonstrate the feasibility of atomic multicast tolerant of malicious process failures without relying on synchrony assumptions. This work has yielded a new protocol for asynchronous atomic multicast, as well as new protocols for reliable multicast (i.e., Byzantine agreement [17]), process group membership, and output voting [23, 24, 22]. In addition, Rampart provides efficient implementations of these protocols to facilitate the construction of high-integrity services in practice.

Rampart facilitates the construction of high-integrity services by relieving the application programmer of the complexities of server replication. That is, our goal is to enable a programmer to build a (non-replicated) application server program and then, with only minor and very simple changes to the server code, use Rampart to replicate it. The primary interfaces provided by Rampart enable (i) application clients to issue requests to the replicated service, which are delivered to all correct application servers in the same order (i.e., atomic multicast) and (ii) application servers to send replies to clients, in a manner that ensures that

the replies delivered to clients are only those sent by correct application servers (i.e., output voting).

This paper gives a brief overview of Rampart and a snapshot of the system at the time of this writing. Most of this paper is concerned with the atomic multicast protocol of Rampart, which is the topic of Section 2. We discuss the output voting protocols of Rampart in Section 3, and the status of Rampart and ongoing work in Section 4. All performance numbers discussed in this paper were obtained in tests among user processes running on a network of moderately loaded SPARCstation 10s over SunOS 4.1.3.

2 Atomic multicast

In our present implementation, atomic multicast from clients to servers is implemented using an atomic multicast protocol from *servers* to servers. This latter protocol, moreover, is implemented using two additional protocols, namely a group membership protocol and a reliable multicast protocol (see Figure 1). These membership, reliable multicast, and atomic multicast protocols are described in Sections 2.1–2.3, respectively, and the extension of this atomic multicast implementation to enable clients to multicast to servers is discussed in Section 2.4. The protocol architecture shown in Figure 1 is similar to that of the Isis system, and indeed Rampart was heavily influenced by Isis and its successor Horus [4]. Rampart distinguishes itself primarily by providing robustness against a broader class of failures than do Isis and Horus, albeit at a significant cost in complexity and performance.

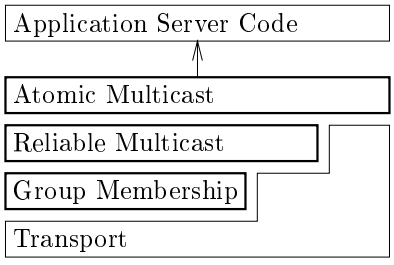


Fig. 1. Atomic multicast implementation (server process)

The protocols pictured in Figure 1 make two main assumptions regarding the operational environment in which they run. First, since they employ digital signature technology (in our present implementation, RSA [25]), they assume that the Rampart instance at each server possesses a private key, which only it knows, and all other servers' corresponding public keys. This initial key distribution can be performed manually by an operator, with the help of a key distribution infrastructure (e.g., [18]), or simply by storing servers' public keys in nonvolatile memory at each server. Similarly, the public key of a new server must be communicated to the other servers before it can be put into operation.

Second, these protocols assume the existence of a message transport facility that provides a point-to-point, reliable, authenticated communication channel between each pair of servers, where by "reliable" we mean that if the sender and destination of a message are correct, then the destination eventually receives the message. Reliability, however, is required only for the liveness of our protocols, not their safety. In our present implementation, this transport facility is provided by the Multicast Transport Service (MUTS) [30]. MUTS implements authenticated point-to-point communication with standard shared-key techniques [31], using shared keys distributed via the public keys described above. MUTS also provides several other facilities (e.g., threads, timers, and synchronization primitives) that have significantly simplified our implementation.

It is also worthwhile to reiterate what Rampart's atomic multicast implementation does *not* assume. Because of our desire to model corruptions by an attacker, it does not make assumptions regarding the types of failures that servers (or clients) can exhibit. That is, our failure model is Byzantine (with message authentication [7]), although the number of such server failures must be limited to a constant fraction of the servers; this is discussed further in Sections 2.1 and 2.2. In addition, our atomic multicast protocol assumes no bounds on message transmission times or clock drifts. That is, the system is *asynchronous*.

2.1 Group membership

The foundation of atomic multicast is a group membership protocol [23]. This protocol provides the abstraction of a *group* of operational servers that may change to reflect the perceived failure or recovery of servers, or the addition of new servers. More precisely, the group membership protocol generates a sequence of *group views*, each of which is a set of server process identifiers, and delivers each view to all members of that view. The membership protocol also provides an interface by which processes

can request additions to or removals from the group. Subject to conditions described below, the membership protocol ensures that if sufficiently many members of a group view request that a member be removed, then that server will eventually be removed from the group (i.e., a view will be created not containing that server), and similarly for requests to add a server to the group. In addition, it ensures that malicious members cannot cause membership changes or prevent needed changes from occurring.

The semantics provided by the membership protocol are contingent on an assumption that fewer than one-third of the members of each group view are faulty. Moreover, since the membership protocol generates agreement on the sequence of group views, it is a type of consensus protocol and thus cannot be guaranteed to be live due to the impossibility result of [10]. In our protocol, this limitation manifests itself in that the creation of a new view can be guaranteed only if there exists a correct group member whose removal is not requested for sufficiently long (and thus who is reachable for sufficiently long) by more than two-thirds of the current group members. Advances on reaching consensus in the presence of crash failures (e.g., [6]) hint that it may be possible to somewhat weaken this requirement; this is a topic for future research.

Changing the group membership (i.e., generating a new group view) in our present implementation is a heavyweight operation. For instance, if RSA keys with 512-bit moduli are used, then the latency of a group membership change is roughly 200 milliseconds in a group of size four (see [23]). However, since experience with current group-oriented systems suggests that membership changes are infrequent for most applications [3], and since we expect this to be the case in the Rampart applications that we currently envision, we anticipate that this cost will not be a limiting factor for most applications. Moreover, since the cost of the group membership protocol is dominated by RSA operations, substantial improvements can be realized by using commercially available, special-purpose hardware to perform these operations [5, 9].

2.2 Reliable multicast

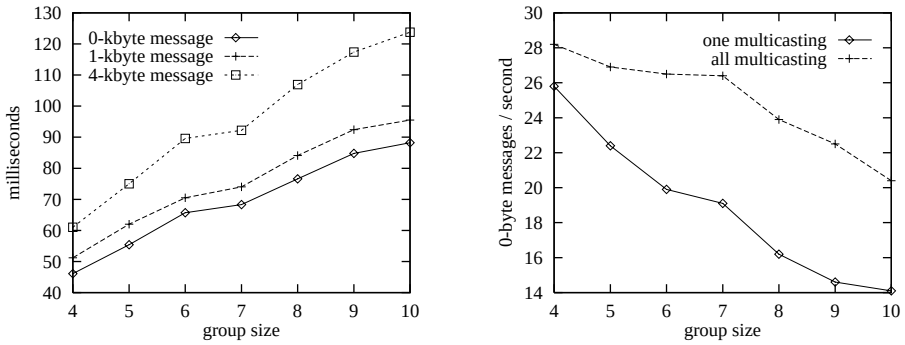
Reliable group multicast is implemented over the membership protocol. The reliable group multicast protocol provides an interface by which group members can multicast messages to the group of servers. The protocol delivers a sequence of *events* to each group member, where each event is either a message that was reliably multicast by a group member or a group view that was received locally from the membership protocol. The reliable multicast protocol delivers group views to each correct

member in the order they are received from the membership protocol. In addition, the set of messages that it delivers between any two consecutive group views is the same at each correct server that is not removed from the group. Thus, this protocol layer implements a *virtually synchronous* process group abstraction as originally defined by the Isis system [3].

Our reliable multicast protocol is detailed in [22]. Like the membership protocol, our reliable multicast protocol relies on the assumption that fewer than one-third of the members of each group view are faulty, due to both its use of the membership protocol and other details of the reliable multicast protocol itself. Moreover, in some cases the progress of the reliable multicast protocol relies on the removal of a member from the group (in which case the removal will be requested by sufficiently many members). Thus, the liveness of our reliable multicast protocol is contingent upon the liveness of the membership protocol.

Since atomic group multicast is implemented using reliable group multicast, it is essential that reliable multicast perform well. As in the group membership protocol, the cost of reliable multicast is dominated by RSA operations. To lessen this cost, our reliable multicast protocol uses "short-term" cryptographic keys that are marginally secure and thus faster to use. To compensate for the use of weak short-term keys, we employ additional protocols to change these keys frequently. For example, we typically use RSA keys with only 300-bit moduli, and change these keys once per hour.[1] Reliable multicast performance for this key size (and in the absence of failures) is shown in Figure 2. Part (a) shows the mean latency of a "multi-RPC", in which a single member reliably multicasts a message (of either 0, 1, or 4 kilobytes) to the group and all group members reply with a null point-to-point message. The latency is the time that elapses at the initiator between initiating the multi-RPC and receiving all replies. Part (b) shows the throughput achieved in tests where either one member ("one multicasting") or each member ("all multicasting") initiated 100 0-byte reliable multicasts to the group. Throughput was computed as the ratio of the total number of reliable multicasts to the time required for them to complete. Despite the use of short-term keys, the cost of reliable multicast is still dominated by RSA operations. Performance could be improved by using smaller short-term keys, at the cost of changing them more often, or by using special-purpose hardware to perform RSA operations.

[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 the largest general RSA modulus factored to date [2] (A. Odlyzko, private communication, May 1994).



(a) multi-RPC latency (b) throughput

Fig. 2. Server-to-group reliable multicast performance

2.3 Atomic multicast

Atomic group multicast is implemented over the reliable group multicast protocol. Like reliable multicast, the atomic multicast protocol provides an interface for members to multicast messages to the group, and it delivers a sequence of events to each member, where each event is a group view or a message. The atomic multicast protocol delivers the same sequence of events to each correct member that is not removed from the group.

Our atomic multicast protocol is similar to those of the Amoeba [16] and Isis [3] systems, which are tolerant to only benign failures. In these systems, there is a designated member of each group view, called the *sequencer*, that determines the order in which atomic multicasts to that view are delivered, using one of two methods. In the first method, called "PB" in [16], the member initiating an atomic multicast does so by sending the message to the sequencer. Upon receiving the message, the sequencer forwards it to the group using a FIFO multicast, and all members deliver atomic multicast messages in the order they are received from the sequencer. In the second method, called "BB", the member initiating the atomic multicast FIFO-multicasts the message, say m, to the group. Upon receiving m, the sequencer FIFO-multicasts a second message naming m as the next atomic multicast message to be delivered. With either method, atomic multicasts are delivered in the same order at all group members because their order is dictated by the sequencer. Techniques to handle the (benign) failure of the sequencer (or any group member) are discussed in [16, 3].

A first step to adapting these protocols to handle malicious members is to replace all FIFO multicasts by (FIFO) *reliable* multicasts using the

protocol of Section 2.2. This modification ensures that all correct members deliver the same sequence of atomic multicasts. However, a corrupt sequencer could still prevent an atomic multicast from being delivered, e.g., by ignoring the message when it is received from the atomic multicast initiator. Such attacks can be countered in the BB protocol: if the correct members do not deliver an atomic multicast message within some timeout period after receiving it by reliable multicast from the atomic multicast initiator, then each correct member requests that the sequencer be removed from the group. Thus, a denial of service attack by the sequencer will result in the sequencer's removal in the BB approach (contingent on the liveness of the membership protocol), whereas it is more difficult to counter such attacks in the PB approach. It is for this reason that the BB approach was initially employed in Rampart [22]. On the other hand, the PB approach provides better performance, because there is only one reliable multicast on the critical path of an atomic multicast (versus two in the BB approach), and because many atomic multicast messages can be communicated to the group in a single reliable multicast from the sequencer.

For these reasons, in Rampart we have moved to an atomic multicast protocol that is a hybrid of the PB and BB approaches described above. During normal operation, the protocol uses the PB approach: for a member to atomically multicast a message, it sends the message to the sequencer. The sequencer collects messages and periodically reliably multicasts a message containing a sequence of messages that constitute the next atomic multicast deliveries. If a member detects that a message it sent to the sequencer is not being included in the sequencer's reliable multicasts, then the member reliably multicasts the message to the group. If the sequencer still does not inform the group when to deliver this message in the atomic multicast delivery sequence, then eventually all correct group members will request that the sequencer be removed. For brevity, we defer further details of the protocol to a forthcoming paper.

An example of the throughput attainable with this protocol (in the absence of failures) is shown in Figure 3. This figure shows the results of tests in which either one non-sequencer member ("one multicasting") or each member ("all multicasting") initiated 1000 0-byte atomic multicasts to the group. The throughput was computed as the ratio of the total number of atomic multicasts to the time required for these atomic multicasts to complete. In these tests, the sequencer issued a reliable multicast approximately every 100 milliseconds. Mean latency is not shown in Figure 3 because this value may bear little relation to the actual la-

tency experienced per atomic multicast: the latency of an atomic multicast varies based upon the time between the sequencer's receipt of the multicast message from the atomic multicast initiator and the sequencer's next reliable multicast. The frequency of the sequencer's reliable multicasts is important to the performance of this protocol; finding optimal such frequencies is a topic of ongoing work. Another area of ongoing work is flow control, the lack of which caused the sequencer to be overwhelmed in the tests of Figure 3 for groups of size greater than seven, resulting in a dramatic decline in throughput.

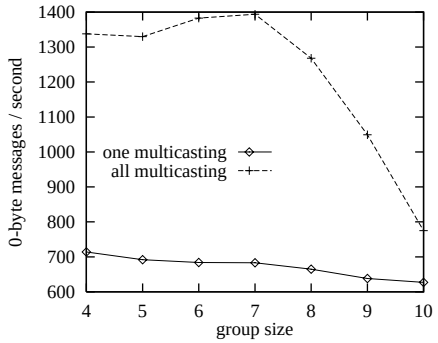


Fig. 3. Server-to-group atomic multicast throughput

2.4 Clients

While so far this section has focused on atomic multicast from servers to servers, ultimately our goal is to enable *clients* to atomically multicast requests to the servers. In our present implementation, a client issues a request to the service by sending it to a single server, which forwards the request by atomically multicasting it to the server group. This approach has several attractive features. First, it does not require the client to know for certain where each of the servers is. That is, the client need only be able to reach one of the correct servers, which it can locate, e.g., by broadcasting a query to which the servers respond or by obtaining a hint for the address of a server from some (not necessarily trusted) source. Second, this approach is very efficient, especially because a server can forward many client requests to the server group with a single atomic group multicast. (This is similar to transaction *batching* as described in [21].) For example, Figure 4 illustrates the throughput of client requests to a service

consisting of four servers in our present implementation. In these tests, each client issued 1000 0-byte requests to the service, and the throughput was computed as the total number of requests divided by the time required for all of them to be delivered to the servers. Each server initiated an atomic multicast to the group (containing perhaps many client requests) approximately every 40 milliseconds, and the sequencer in the atomic multicast protocol issued reliable multicasts approximately every 100 milliseconds.

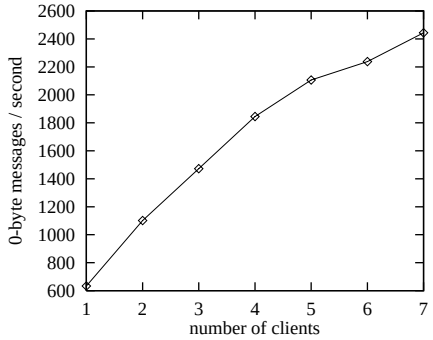


Fig. 4. Client-to-group atomic multicast throughput

The main risk of this approach is that the server to which a client sends its requests may be corrupted. In this case, the client could be denied service or, if the client's requests are not authenticated by the correct servers, the corrupt server could undetectably alter the client's requests in transit. The latter attack can be detected by the correct servers authenticating client requests using standard cryptographic techniques at the application layer [31]. If a denial of service occurs, then the client must locate another server to which to send its requests.

3 Output voting

Application servers communicate outputs to clients via output voting protocols, which ensure that the replies delivered to clients are only those sent by correct servers (see Section 1). We have implemented two types of output voting protocols in Rampart. In the first, the output from each server is sent to the client that requested it, and the client performs output voting in the standard way [26]. A disadvantage of this approach is that the client must be able to identify and authenticate each server

individually. (Note that this set of servers that the client must authenticate is different from, but encompasses, the potentially more dynamic server *group* defined by the group membership protocol for the purposes of reliable and atomic multicast; see Section 2.1.)

The second protocol is novel in that output voting is performed at the servers and is transparent to clients [24]. That is, for a client to verify an output from the service with this protocol, the client need not know how many servers there are or how to authenticate individual servers. Instead, it need only possess one public key for the service (not to be confused with the public keys of individual servers) and verify one reply from the service, just as if the service were not replicated. This is achieved by using cryptographic methods to "split" the private key corresponding to the service's public key among the servers, so that the cooperation of sufficiently many servers is required to sign a response [8]. This protocol offers advantages over the usual approach to output voting if there is no trustworthy way for clients to identify or authenticate individual servers.

The main disadvantage of this latter approach is that the cryptographic techniques that it employs are costly: if the RSA public key of the service contains a 512-bit modulus, then the latency of output voting with four servers is roughly 200 milliseconds. Moreover, this key may not be amenable to frequent change to minimize its size (as in Section 2.2), because this would force clients to stay current with these key changes. Thus, for many applications this approach to output voting may require special-purpose hardware at the servers for performing RSA operations, and in the absence of such hardware, the more standard technique of output voting may be preferable.

4 Status and ongoing work

An initial version of Rampart is nearing completion at the time of this writing. In particular, prototypes of the group membership, reliable multicast, atomic multicast, and output voting protocols are fully operational. Preliminary performance numbers for these protocols indicate that Rampart is likely to provide performance that suffices for many of the applications that initially motivated it. Nevertheless, they also indicate a high cost for tolerating malicious processes; e.g., the latency of our reliable and atomic multicast protocols in our present implementation compare poorly to those of some published protocols that tolerate only benign failures.

Our current focus is optimizing our protocols and completing the implementation of other mechanisms in Rampart, such as protocols for new

servers to be integrated into a service. In conjunction, we are prototyping services with Rampart (e.g., [11]) that, while of interest in their own right, demonstrate the toolkit's utility and provide insight into where the toolkit is lacking.

Another area of ongoing research is to extend the Rampart protocols to provide support for a wider range of security technologies. For example, many theoretical techniques to achieve secure distributed computation (see [12] for a survey) use reliable multicast as a basic building block, and thus our reliable multicast protocol might facilitate the realization of these techniques in practice. Similarly, our group membership protocol has potential applications to intrusion confinement and maintaining membership information in groups of processes that are mutually distrusting.

Acknowledgements

We thank the workshop participants—especially Ken Birman, Danny Dolev, Vassos Hadzilacos, Farnam Jahanian, Gérard Le Lann, Keith Marzullo, André Schiper, Rick Schlichting, Sam Toueg, and Robbert van Renesse—for discussions that clarified many theoretical and practical issues surrounding Rampart. We also thank Matthew Franklin, Stuart Stubblebine, Sandra Thuel, Rebecca Wright, and the anonymous referees for commenting on drafts of this paper.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.
2. 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.
3. K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug. 1991.
4. K. P. Birman and R. van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, California, 1994.
5. E. Brickell. A survey of hardware implementations of RSA. In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89 Proceedings* (Lecture Notes in Computer Science 435), pages 368–370. Springer-Verlag, 1990.
6. T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 325–340, Aug. 1991.
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. Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91 Proceedings* (Lecture Notes in Computer Science 576), pages 457–469. Springer-Verlag, 1992.
9. S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology—EUROCRYPT '90 Proceedings* (Lecture Notes in Computer Science 473), pages 230–244. Springer-Verlag, 1991.
10. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.
11. M. K. Franklin and M. K. Reiter. The design and implementation of a secure auction service. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, May 1995. To appear.
12. 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.
13. L. Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, Apr. 1989.
14. L. Gong. Securely replicating authentication services. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 85–91, 1989.
15. M. P. Herlihy and J. D. Tygar. How to make replicated data secure. In C. Pomerance, editor, *Advances in Cryptology—CRYPTO '87 Proceedings* (Lecture Notes in Computer Science 293), pages 379–391. Springer-Verlag, 1988.
16. M. F. Kaashoek. *Group Communication in Distributed Computer Systems.* PhD thesis, Vrije Universiteit, The Netherlands, 1992.
17. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
18. B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.
19. 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, Jan. 1990.
20. L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug. 1989.
21. F. M. Pittelli and H. Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems*, 7(1):25–60, Feb. 1989.
22. 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, Nov. 1994.
23. 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.
24. 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.
25. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.

26. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.
27. 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.
28. J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, Feb. 1988.
29. R. Turn and J. Habibi. On the interactions of security and fault-tolerance. In *Proceedings of the 9th NBS/NCSC National Computer Security Conference*, pages 138–142, Sept. 1986.
30. R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX Microkernels and Other Kernel Architectures Workshop*, Apr. 1992.
31. V. L. Voydock and S. T. Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15(2):135–171, June 1983.