Cryptographic Support for Fault-Tolerant Distributed Computing[*]

Yaron Minsky[†] Robbert van Renesse [‡] Fred B. Schneider[†]
Scott D. Stoller[†]

Department of Computer Science
Cornell University
Ithaca, New York 14853, U.S.A

July 6, 1996

1 Introduction

Mobile processes, or *agents*, have been proposed for a variety of applications in the Internet and other large distributed systems. But little work has been directed at operating-system support for agents. This paper discusses one aspect of the problem—implementing fault-tolerance without specialized hardware.[1]

In traditional client-server settings, a central and trusted host may send all messages and receive all replies, thereby implementing a star-shaped communications pattern. In contrast, an agent can execute autonomously at a succession of remote sites without returning to the host that launched it. Thus, computations structured using agents may consume less network-bandwidth in performing tasks that involve multiple hosts. Moreover, for some settings, it is unrealistic to presume the existence of a central host that remains connected to the network—mobile computing and wireless networks are obvious examples.

In an open distributed system, agents comprising an application must not only survive (possibly malicious) failures of the hosts they visit, but they must also be resilient to the potentially hostile actions of other hosts. Correctness of a computation should depend only on hosts that would be visited in a failure-free run. We assume that faulty hosts produce erroneous messages, that they can masquerade as other faulty hosts, but that they cannot assume the identities and do not have access to secrets of non-faulty hosts.

Replication and voting are necessary to survive malicious behavior by visited hosts. However, faulty hosts that are not visited by agents can confound a naive replica-management scheme by spoofing. With this in mind, we have been investigating protocols for replication and voting in a family of applications. Our protocols use cryptographic techniques in novel ways. Furthermore, our experiments reveal that fast (correct) hosts can mask delays caused by slow ones, so replication actually speeds up some applications.

Section 2 characterizes the family of applications treated in this paper. Section 3 describes experiments we ran to explore performance implications of replication and voting in this setting. The role of cryptographic techniques in our protocols is discussed in section 4. Section 5 contains our conclusions.

[*]Contact author: Fred B. Schneider, Department of Computer Science, Upson Hall, Cornell University, Ithaca, New York 14853, U.S.A. Email: fbs@cs.cornell.edu

[†]Supported in part by ARPA/NSF Grant No. CCR-9014363, NASA/ARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198.

[‡]Supported in part by ARPA/ONR contract N00014-92-J-1866.

[1]In contrast, [CGH+95] gives solutions that require secure co-processors.

2 A Fault-tolerant Pipeline

A simple agent-computation might visit a succession of hosts, ultimately delivering its results to an actuator (which may be the same host as the source). The difficulties that arise in making such a computation fault-tolerant are typical of those associated with more complex agent-computations. Therefore, we have taken this simple computation as a starting point for our investigations.

The agent-computations of interest can be viewed as a pipeline. See Figure 1. Nodes represent hosts, and edges represent movement of an agent from one host to another. Each node corresponds to a *stage* of the pipeline. S is the *source* of the pipeline; A is the *actuator*.

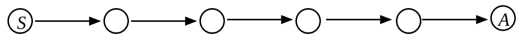

Figure 1: A simple agent computation

This computation is not fault-tolerant. The correctness of a stage depends on the correctness of its predecessor, so a single malicious failure can propagate to the actuator. Moreover, even if there are no faulty hosts in the pipeline, some other (malicious) host could disrupt the computation by sending an agent to the actuator first.

One step towards achieving fault-tolerance is replication of each stage. Assume execution of each stage is deterministic,[2] but the components of each stage are not known *a priori* (because, say, they depend on results computed at previous stages). A node p in stage i takes as its input the majority of the inputs it receives from the nodes comprising stage $i-1$. And, p sends its output to all of the nodes that it determines comprise stage $i+1$. Such a fault-tolerant execution is illustrated by Figure 2.

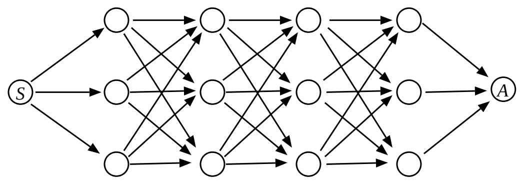

Figure 2: Replicated agent computation with voting

Notice that the computation in Figure 2 tolerates more failures than an architecture where the only voting occured just before the actuator. The voting at each stage makes it possible for the computation to *heal* by limiting the impact of a faulty host in one stage on hosts in subsequent stages. More precisely, in the architecture of Figure 2, it is possible to tolerate faulty values from a minority of the replicas in each stage.

3 Voting Performance Issues

Voting has performance costs. First, there is an intrinsic cost associated with sending, receiving, and comparing votes. Second, there is a synchronization delay—a voter must wait until it has received a majority of identical values before it can emit an output.

Voting also has performance benefits in some situations. This is because voting does not force the computation to wait for the slowest correct agent; a voter need only wait until it has received a majority of

[2]This determinacy assumption can be relaxed somewhat without fundamentally affecting the solution.

identical values. It is therefore the median agent, rather than the slowest, that determines the overall speed of the computation. Thus, voters tolerate slow as well as faulty agents.

To further explore these performance issues, we ran some experiments. The system we measured consisted of 14 SUN SparcStation 20 computers connected by a 10 Mbit Ethernet using UDP/IP for communication. An agent moving from processor to processor was simulated by sending a message between these processors. In the experiments, we looked at the behavior for 1, 3, 5 and 7 replicas.

Our first experiment examined the costs of voting in the case that host speeds are reasonably uniform. We were interested in how synchronization delay could be amortized by voting less frequently. Rather than voting at the end of each stage, agents in this experiment visited a sequence of N hosts before voting. Figure 3 is a graph of the average time per host visit when N ranges from 1 to 50.[3] We found pronounced improvements as N advanced from 1 to 10; for N greater than 10, the further improvements were not significant.

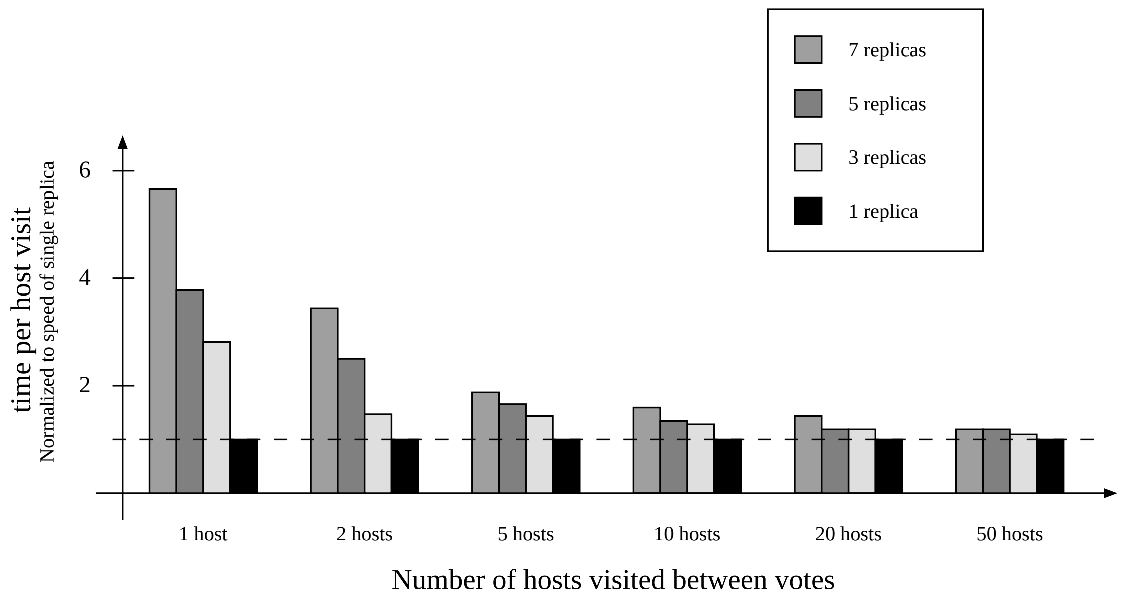

Figure 3: Voting performance with various voting frequencies

One might fear that infrequent voting would be costly because of the resulting high synchronization delay. When voting is rare, replica completion times drift apart, so the synchronization delay increases. However, a voter need only wait for a (correct) majority, so a vote-delimited stage will complete as soon as the median correct replica votes. Therefore, the completion time for a replicated computation that votes infrequently should approximate the completion time when there is a single replica. The data in Figure 3 shows exactly this behavior.

Voting can even lead to a replicated computation being faster than the corresponding non-replicated one. Suppose there is a small probability that any given host will be slow. Over a sufficiently long non-replicated execution, an agent is bound to encounter a slow host; the computation will be slowed accordingly. However, with replication and periodic voting, it is likely that a majority of the agents reaching a voter will have encountered no slow hosts. Since the voter only waits for this majority, the replicated system's execution

[3]The graph reports averages from runs of 300 rounds. The time spent per host had a variance of .1%.

time will be independent of the speed of the slow hosts.

Figure 4 shows the performance of an agent that voted every five moves. We set the probability of encountering a slow host to 2%, and such a host was 350 times slower then a normal host. The unreplicated computation felt the full effect of the impaired hosts, experiencing a sixfold slowdown. The slowdown was greatly reduced in the presence of any replication, dropping to 20% for three replicas. The addition of replicas further reduced the slowdown, which became negligible with seven replicas.

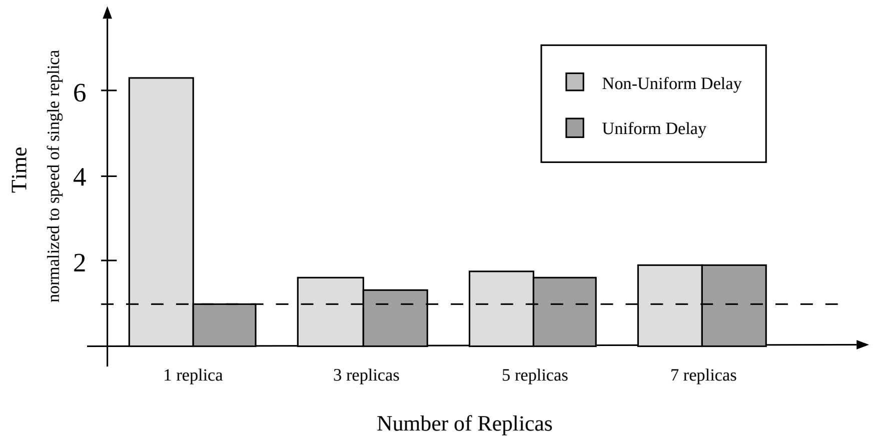

Figure 4: Voting performance with uniform and non-uniform delays

4 Cryptographic Support

The computation of Figure 2 should tolerate one malicious host per stage of the pipeline. It doesn't. Any two faulty hosts could claim to be in the last stage and foist a majority of bogus agents on the actuator. These problems are avoided if the actuator can detect and ignore such bogus agents. This could be accomplished by having agents carry a *privilege* from the source to the actuator.

The privilege can be encoded as a secret initially known only to the source and the actuator. It is necessary to defend against two attacks. First, a malicious host might misuse the secret and launch an arbitrary agent to the actuator. Second, a malicious host could destroy the secret, making it impossible for the remaining (correct) agents to deliver a result.

To simplify the discussion, we start with a scheme that prevents misuse of the secret by a malicious host. This scheme ensures that, provided that a majority of replicas visit only correct hosts, no host except the source and the actuator will learn the secret.

Agent replicas cannot simply carry copies of the secret, since the secret could then be stolen by any faulty host visited by a replica. It is tempting to circumvent this problem by the use of an (n, k) threshold secret sharing scheme [Sha79] to share the secret embodying the privilege.[4] In a system having $2k - 1$ replicas, the source would create fragments using a $(2k - 1, k)$ threshold scheme, and send a different fragment to each of the hosts in the first stage. Each of these hosts would then forward its fragment to a different host in the next stage, etc.

[4]In an (n,k) threshold scheme, a secret is divided into n *fragments*, where possession of any k fragments will reveal the secret, but possession of fewer fragments reveals nothing.

This protocol fails, however, due to the voting at intermediate stages of the pipeline. The voting should ensure that the faulty hosts encountered before a vote cannot combine with faulty hosts encountered after the vote to corrupt the computation. However, in the scheme just described, minorities before and after the vote can steal different subsets of the secret fragments. If together they hold a majority of the secret fragments, then the faulty hosts can collude to reconstruct the secret.

One way to stop collusion between hosts separated by a vote is to redivide the secret fragments at each vote. The following outlines a protocol based on this insight. For a system with $2k-1$ replicas:

- The source divides the secret into $2k-1$ fragments and sends each fragment to one of the hosts of the first stage.
- A host in stage i takes the fragments it receives, concatenates them, and divides that into $2k-1$ fragments using a $(2k-1, k)$ threshold scheme. Each fragment is then sent to a different host in stage $i+1$.
- The actuator uses the threshold scheme (backwards) and recovers the original secret.

This protocol is inefficient because secret sharing schemes require that each fragment be the same size as the original secret. Thus, messages get longer at every stage of the pipeline. In fact, the message size is multiplied by $2k-1$ at every stage.

We have developed two protocols that do not suffer from this exponential blowup in message size. In one protocol, message size grows linearly with the number of pipeline stages and the number of replicas. In the other, message size remains constant but an initialization phase is required. The first scheme uses chains of authentication (instead of a secret privilege) to prevent masquerading. The second scheme renews [Jar95] the secret after each round, making it impossible for fragments from before a vote to be used together with fragments constructed after that vote.

To address the second attack described above — destruction of the secret by a faulty host — it is necessary to replace the secret sharing in the protocols just described with verifiable secret sharing (VSS).[BOGW88, FR95] VSS schemes allow for correct reconstruction of a secret (or uniform discovery that no secret was distributed) even when hosts, including the source, are faulty.

5 Conclusion

This paper describes preliminary experiences with implementing fault-tolerance for agent computations. We found that replication and voting do not suffice. Cryptographic support was required. We also found that delays introduced by voting could be rendered insignificant by making voting less frequent. Moreover, in some cases, replication and voting actually improve performance by ensuring that slow hosts do not impede progress.

Cryptographic techniques have been used before in distributed protocols [LSP82, GLR95, CR93]. Secret sharing, in particular, has been employed for asynchronous Byzantine agreement [CR93], secure auctioning [FR95] and secure function evaluation [BOGW88]. All of this work, however, depends on computations being immobile. In the work we report here, secret sharing takes on a new role in facilitating mobile processes by providing a form of distributed authentication. It is our belief that this new type of authentication is fundamental to supporting integrity of mobile agents in distributed systems.

References

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Widgerson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *ACM Symposium on Theory of Computing*, pages 1–10, 1988.

[CGH$^+$95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Paris, and G. Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications 2*, 5:34–49, October 1995.

[CR93] Ran Canetti and Tal Rabin. Optimal asynchronous Byzantine agreement. *25th Symposium on Theory of Computing*, pages 42–51, 1993.

[FR95] Matthew K. Franklin and Michael K. Reiter. The design and implementation of a secure auction service. *IEEE Symposium on Security and Privacy*, 1995.

[GLR95] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: observations and applications in tolerating hybrid and link faults. *Dependable Computing for Critical Applications 5*, pages 79–90, September 1995. Champaign, Ill.

[Jar95] Stanislaw Jarecki. Proactive secret sharing and public key cryptosystems. Master's thesis, MIT, September 1995.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM TOPLAS 4*, 3:382–401, July 1982.

[Sha79] Adi Shamir. How to share a secret. *CACM*, 22:612–613, November 1979.