Towards Low Latency State Machine Replication for Uncivil Wide-area Networks

Yanhua Mao
CSE, UC San Diego
San Diego, CA - USA
maoyanhua@cs.ucsd.edu

Flavio P. Junqueira
Yahoo! Research
Barcelona, Spain
fpj@yahoo-inc.com

Keith Marzullo
CSE, UC San Diego
San Diego, CA - USA
marzullo@cs.ucsd.edu

Abstract

We consider the problem of building state machines in a multi-site environment in which there is lack of trust between sites, but not within a site. This system model recognizes the fact that if a server is attacked, then there are larger issues at play than simply masking the failure of the server. We describe the design principles of a low-latency Byzantine state machine protocol, called RAM, for this system model. RAM also makes judicious use of attested append-only memory (A2M) and uses a rotating leader design to further reduce latency.

1. Introduction

State machine replication is a popular foundation for constructing highly available distributed services. Byzantine fault tolerance (BFT) is the most used approach for adapting state machine replication to address service fault tolerance against adversarial attacks. The first step towards making this approach practical was PBFT [5], which provides high throughput BFT state machine replication.

We consider the problem of building state machines in a multi-site environment in which there is lack of trust between sites. Given a total of n sites, we have each site supply one state machine replica, and assume that no more than f servers can exhibit Byzantine behavior. If a server behaves in a Byzantine manner because of a fault in the service software, then the problem is widespread and hard to contain: it would need to be detected and fixed using debugging techniques. If the server is Byzantine because of a failure in the site's system administration, then the problem is quite severe for that site but less so for the other sites: masking the server's faulty behavior is short-sighted. Instead, the system administrators need to understand how widespread the damage is and restore the site's integrity.

We call this model of site behavior *Mutually Suspicious Domains*, or *MSD* for short. In such an environment, a client can trust the server in its site since there are more general mechanisms needed for dealing with compromised servers and failures of system administration in that site. Servers in other sites, however, are treated as being possibly *uncivil* [7], *i.e.*, they may not follow the protocol.

We investigate BFT state machine replication in MSD environments. While most previous work have focused on providing high throughput [5, 7, 12], our goal is low latency. We have this goal because low latency is difficult to achieve in real wide-area replicated systems. Throughput increases over time because of higher performance hardware and higher bandwidth communication channels; reducing latency, however, requires redesigning protocols.

As an example, the PNUTS [8] platform replicates data in a wide-area system and has low latency as a primary goal. With PNUTS, data replicas reside in several continents, and users access data locally. For availability, data is replicated across different sites. The latency for guaranteeing serializability across sites, however, is too high to meet the user expectation. Consequently, the designers of PNUTS have opted for relaxed per-record consistency that is weaker than serializability but stronger than eventual consistency.

Our plan is to use an approach similar to what we used in Mencius [15], which is a wide-area state machine replication protocol for crash failures. As with Mencius, replicas take turns as the leader and propose client requests in their turns. Doing so reduces latency because it enables client requests to be proposed directly by their local replicas.

The result of our investigation is the initial design of a protocol that we call RAM. RAM leverages our MSD model, a rotating leader scheme, and attested append-only memory (A2M) to reduce latency in multi-site environments. Throughout this paper, we argue that the properties of RAM lead to a low-latency protocol in multi-site environments and that it is a practical protocol.

2. PBFT revisited

To implement a replicated state machine, a group of servers run an unbounded number of instances of *consensus* to agree on a sequence of client requests to execute. Masking Byzantine failures is inherently harder than masking crash failures. The result is that BFT protocols typically have higher latency than their crash-failure counterparts. In this section, we identify the underling cause of the higher latency by comparing two protocols: Paxos [13] and PBFT [5]. Paxos is a crash failure consensus protocol that tolerates f failures with $2f + 1$ replicas. PBFT implements consensus that tolerates f Byzantine failures with $3f + 1$ replicas. For simplicity, we assume $n = 2f + 1$ for Paxos and $n = 3f + 1$ for PBFT in the following sections.

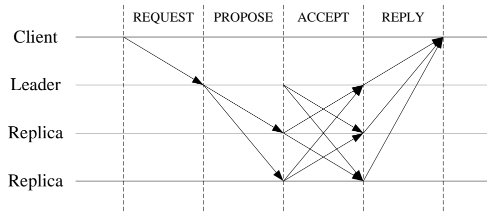

Figure 1: The message flow of Paxos in failure free runs.

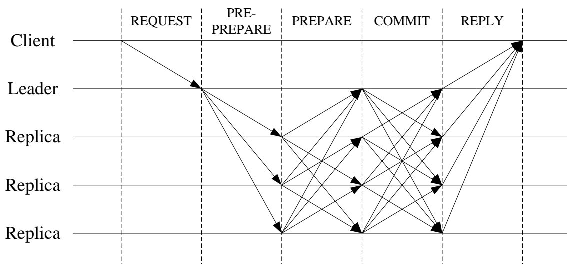

Figure 2: The message flow of PBFT in failure free runs.

Paxos is a leader based protocol. Figure 1 shows the message flow of Paxos in failure-free executions. At a high level: (a) it starts when the client sends its request to the leader using REQUEST message; (b) the leader then assigns the request with a consensus sequence number and proposes it by broadcasting PROPOSE messages; (c) if there is no failure or concurrent leader, the server will accept the proposal by exchanging ACCEPT messages; (d) a server knows that consensus has been reached if a majority of the servers have accepted it. It then executes the request and sends a REPLY back to the client; (e) The client learns the outcome of the request as soon as it receives the first reply.

PBFT is also a leader (primary) based protocol. Figure 2 shows the message flow of PBFT in failure-free executions. At a high level: (a) it starts when the client sends its request to the leader using REQUEST message; (b) the leader then assigns the request with a consensus sequence number and disseminates this assignment by broadcasting PRE-PREPARE messages; (c) after that, the servers exchange PREPARE messages to agree on the assignment the leader has proposed; (d) if $2f + 1$ matching PREPARE messages are obtained, the servers then exchange COMMIT messages to reach consensus, *i.e.*, to guarantee that even if a leader (view) change occurs the new leader must propose the same value for this sequence number; (e) this is established when a server gathers $2f + 1$ matching COMMIT messages. Then it can execute the request and send a REPLY back to the client; (f) the client learns the outcome of the request once it receives $f + 1$ matching REPLIES.

When running Paxos in multi-site systems, the latency of a client request is two wide-area communication steps when the client is co-located with the leader and three steps otherwise. PBFT requires two more steps than Paxos in both cases. This is a result of the following two observations. (1) With PBFT, to prevent a faulty leader from proposing inconsistent values to different servers, two steps (PRE-PREPAREand PREPARE) are required to propose a request. However, only one step (PROPOSE) is required with Paxos since it assumes crash failures. (2) A Paxos client learns the outcome of the request as soon as it hears it from its local server, while a PBFT client must wait for $f + 1$ matching REPLIES at least f of which must cross the wide-area network. In the next section, we explore techniques to reduce latency for PBFT in multi-site systems.

3. Reducing Latency for PBFT

3.1. Mutually Suspicious Domains

In the Byzantine model, there is no trust between pairs of processes, and both clients and correct servers have to be aware of Byzantine servers. Applying the Byzantine model to a multi-site system means that the clients do not even trust their local servers. However, in many practical systems, the clients and their local servers share fate: (1) Clients may rely solely on their local servers to make progress. (2) Client and server machines in the same site are more likely to share vulnerabilities as they are in the same administrative domain. (3) When a local server is compromised, instead of simply masking the failure, it is often the case that administrative actions are required to recover the server and the clients rely on it. We therefore propose the following practical failure model.

Mutually Suspicious Domains (MSD): We model each site as an independent communication domain. While there is trust between the server and clients within a domain, we assume no trust for inter-domain communication, *i.e.*, a domain must protect itself from possible uncivil behavior from other domains. Assuming MSD gives us two advantages in term of reducing latency for BFT protocols:

Local replies: By assuming MSD and trusting its local server, a client can immediately learn the outcome of a request when it receives the REPLY from its local server, therefore reducing its latency by one wide-area communication step.

Local reads: Certain read-only requests that do not require linearizability can now be executed locally without going through wide-area communication.

Note that local reads may violate real-time precedence ordering thus making the history of operation not linearizable [11]. However, serializability is still preserved: clients only perceive that requests are not executing atomically if they communicate through channels that are external to the replicated state machine. If clients only communicate through the state machine, then they perceive the same order of state changes.

3.2. Rotating leader design

PBFT, as well as other more recent BFT protocols [7,12], relies on a single leader to propose requests. This makes it possible for a Byzantine leader to mount performance attacks by, for example, delaying to propose requests from other sites [2]. With MSD, clients can trust their local servers. Consequently, a server proposes requests on behalf of its local clients. This way, the clients do not have to worry about the server being unfair. The result is a rotating leader scheme as defined below, similar to that of Mencius [15], which is a rotating leader version of Paxos.

Simple consensus: Instead of consensus, we now run an unbounded sequence of *simple consensus* instances. In each instance, one distinguished server is assigned as the *coordinator* (the default leader), and all other servers are called *followers*. Simple consensus requires that only the coordinator proposes either a client request (*suggest*) or a *no-op* (*skip*), and followers only propose *no-op* (*revoke*).

Rotating leader: The consensus sequence space is partitioned so that the servers take turns to be the coordinator. The simplest way is to assign coordinators in a round-robin fashion.

In addition to reducing the risk of being treated unfairly by a server in a remote site, rotating the leader allows the REQUEST message to be sent as a local area message. Doing this reduces the latency by one wide-area communication step as compared to PBFT when clients are not co-located with the leader. When there are no concurrent client requests, this reduces the latency observed by the client by one step. As with Mencius, concurrent requests may cause the *delayed commit* problem and increase the latency by up to one step [1]. This extra latency can be reduced by allowing *out-of-order commit*, the same technique used in Mencius, to safely commit commutable requests at different servers in different orders.

[1]As showed in [15], delayed commit can be up to two communication steps when the servers only send ACCEPT messages to a distinguished server. When the servers broadcast ACCEPT messages, the upper-bound is reduced to one step.

3.3. A2M for identical Byzantine failure

While it only takes Paxos one round (PROPOSE) for the servers to know the value proposed by the leader, it takes PBFT two rounds (PRE-PREPARE and PREPARE) to agree on the leader's proposal. The extra delay is necessary to expose inconsistent proposals due to a Byzantine leader, *e.g.*, a Byzantine leader may propose some request v to one server and *no-op* to another. By running a flooding protocol using the extra PREPARE phase, PBFT is able to simulate identical Byzantine failure [4]: if a server has prepared a value from the leader, all other servers will prepare either the same value or no value.

Attested append-only memory (A2M): A2M [6] has been proposed to design BFT protocols that tolerates f failures with only $2f + 1$ replicas, which requires recording all outgoing messages into A2M and making sure they are consistent with respect to the semantics of the BFT protocol. We, however, propose to use A2M in a limited basis to implement identical Byzantine failure with only one wide-area message delay. Each server is paired with a local A2M, a trusted third-party. Whenever a server proposes a value (either *no-op* or a client request) to a consensus instance, it asks its local A2M to digitally sign the proposal. The A2M will record all pairs of values and sequence numbers it has signed [2] and will only sign a proposal if it is consistent with previous records or has not seen the sequence number before: this way a server cannot have the A2M signing inconsistent proposals.

We can now reduce PBFT's proposing phase from two steps to just one step if each server's A2M can be trusted, an assumption that is only possible if the A2M involved is simple enough to be implemented correctly without vulnerabilities. This assumption might hold in practice because the logic that determines consistency is very simple and is independent of the specific protocol that uses it.

While a faulty A2M can render the replicated state machine to an unsafe state, the damage it can cause is limited because inconsistent proposals can be easily detected and proved by exchanging the signed proposals. Once proven faulty, a server (and its A2M) can be excluded from any future proposal. Also, the state machine state can be rolled back and repaired when suitable.

Finally, there are various ways to implement A2M, each with different guarantees. Chun *et al.* have proposed both a software approach with a separate process on each of the sever machine and a hardware approach with specialized

[2]In practice, only the most recent suggestion and skips need to be recorded because a server is not supposed to leave any gaps in the consensus sequence space.

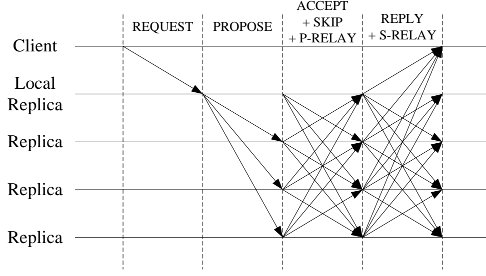

Figure 3: The message flow of RAM in failure free runs.

Message	Meaning
REQUEST	A Client sends a request to a replica
PROPOSE	The coordinator proposes a client request (suggest)
ACCEPT	A Replica accepts the proposal
SKIP	The coordinator proposes *no-op* (skip)
P-RELAY	A replica relays the PROPOSE messages
REPLY	A replica sends the result of a request to the client
S-RELAY	A replica relays the SKIP messages

Table 1: Summary of messages in RAM.

devices [6]. Another way to implement A2M is to implement a replicated service with a separate set of servers at each site. Extra care can be taken with the A2M service to secure it against various attacks.

4. Building Protocol RAM

We have explained three techniques to reduce latency for PBFT. In this section we further explain the high level principle of protocol *RAM* (which stands for **R**otating leader, **A**2M and **M**SD) and the issues that arise in making it efficient and practical. Figure 3 shows the message flow of RAM in failure free executions and Table 1 summarizes the meaning of the messages. Note that PROPOSE, SKIP, P-RELAY and S-RELAY are digitally signed by A2M; all other messages can be either signed by the sending replica or authenticated using Message Authentication Code (MAC).

4.1. RAM: a state machine with BP-A2M

We introduce BP-A2M (which stands for Byzantine Paxos with A2M) to solve each instance of simple consensus in RAM. A server with BP-A2M may invoke one of the three following actions based on its role:

Suggest: The coordinator may *suggest* by proposing a client request. It does this by asking its local A2M to sign this proposal and broadcasting the signed PROPOSE message. Upon receiving the PROPOSE message for the first time, a server accepts it if it has not promised not to do so. It then broadcasts an ACCEPT message and relays the PROPOSE message using P-RELAY. A server learns the outcome when it receives $2f+1$ matching ACCEPT messages.

Skip: The coordinator may *skip* by proposing *no-op*. It does this by asking its local A2M to sign the SKIP message and broadcasting it. Upon receiving the SKIP message for the first time, a server relay the message using S-RELAY to all other servers and learns *no-op* has been chosen because *no-op* is the only possible outcome: the coordinator has proposed *no-op* and all the other servers can only propose *no-op* by the definition of simple consensus.

Revoke: For some protocol parameter α ($1 \leq \alpha \leq f+1$), if $f+\alpha$ servers suspect the current leader is faulty, a new leader is elected to revoke the suspected coordinator. The new leader does this by polling all the other servers and gathering a *progress certificate*. If the progress certificate indicates that some value v might have been chosen, the new leader proposes v to reach consensus, otherwise it proposes *no-op*. It accomplishes this by following the same execution path of suggestions. Note that $\alpha \geq 1$ ensures at least one correct server suspects the current leader. $\alpha \leq f+1$ because up to f servers may be faulty. Choosing a large α reduces the probability of unnecessary revocation caused by network jitter (see section 4.2 for a more detailed discussion).

Note that servers are required to relay PROPOSE and SKIP messages with P-RELAY and S-RELAY messages respectively, because Byzantine servers can omit messages to up to $f+\alpha-1$ correct servers without triggering revocation. While the relay is not necessary during failure-free runs, it ensures that if a correct server learns the outcome, then all correct servers learn the outcome after one round even if some of the servers are Byzantine. During failure-free runs, it takes two rounds to learn a suggestion and one round to learn a skip. During uncivil runs, up to one additional round may be needed for all the servers to learn the outcome.

In failure-free executions, a RAM server suggests its client requests immediately to its *index*, *i.e.*, the next available instance it coordinates. A server is also required to skip its turns when other servers are consuming consensus instance faster, preventing gaps in the consensus sequence. SKIP and P-RELAY messages are piggybacked in the ACCEPT messages for efficiency. Multiple S-RELAY messages are also batched together to reduce the overhead of relaying. When a server has crashed, is slow, or behaves suspiciously, it is revoked to allow other servers to make progress. Revocation is issued in advanced of correct servers' index to prevent the lengthy revocation process from slowing down other servers. Revocation is also done in large block size at a time to amortize the cost. We discuss revocation strategies in more detail in section 4.2.

4.2. Revocation and failure detection

As a result of using the rotating leader design, both Mencius and RAM require eventually accurate failure detector to ensure the liveness of the protocols [15]. While it is not

difficult to implement this class of failure detectors in practice, unavoidable false suspicions come with any practical implementation: it is impossible in an asynchronous system to determine if an unresponsive server has failed or is just slow. When false suspicion happens with Mencius, a server can resume action by setting its index to be greater than the last instance it has been revoked. By suggesting a value to the new index, the falsely suspected server makes other servers update their indexes and skip their unused turns. This quickly synchronizes the indexes of servers.

This strategy, however, should not be applied with RAM: a Byzantine server can abuse it to get out of the revocation and continue to behave maliciously. Not being able to suggest requests on behalf of its clients, a falsely suspected server can either wait the revocation period out or forward its requests to other servers expecting that a correct server proposes them on its behalf. Considering rational behavior [1], this mechanism gives incentive to a rational server to avoid being revoked, *i.e.*, to act according to the protocol and act in a timely fashion.

With the new revocation mechanism, a falsely suspected server can be excluded for action because of occasional network jitter and can not recover quickly. This means that, in addition to eventual accuracy, the underlying failure detector should produce as few false positives as possible. We argue that instead of outputting a binary value (either suspected or not) for each server, a failure detector should output three values: correct, suspicious, or faulty.

A server is believed to be correct if it is timely and no misbehavior is detected. A server is faulty if definite misbehavior is detected, *e.g.*, proposing inconsistent requests to the same consensus instance due to faulty A2M. This class of failure should result in repeated revocation in the future until the faulty server has been repaired and the revocation is lifted through human intervention. Finally, a server is believed to be suspicious if it is not acting fairly or in a timely fashion, for example, failing to batch when it has high client load, failing to accept suggestions from other servers in time, or failing to skip its turns in time. Such a conduct is classified as suspicious because in practice it is difficult to tell whether it is because of network jitter or deliberate step of the server. Ideally, a practical failure detection implementation has a low false positive rate when such behavior is caused by network jitter. Since we require at least $f + \alpha$ servers to revoke a suspected server, having a larger α makes revocation triggered by network jitter less likely: up to $\alpha - 1$ correct servers can experience untimely behavior without triggering revocation even in the presence of f Byzantine servers. Finally, small values for the revocation block size should be used for occasional misbehavior caused by network jitter and large values should be used for repeated deliberate untimely actions. We plan to implement this property by tracking the frequency of each server's misbehavior and choose larger block size for more frequently misbehaved server.

4.3. Limiting damage

In this section we explain why a Byzantine server with RAM can only cause limited increase in latency to requests proposed by other servers without risking itself being suspected by $f + \alpha$ servers and hence being revoked. A RAM server participates in the state machine in two ways: it acts both as the *proposer* for instances coordinated by itself and as the *acceptor* for instances coordinated by other servers. A Byzantine server can only cause a bounded increase in latency because:

(1) As an acceptor, a Byzantine server can not keep requests proposed by a correct server from being chosen by not accepting its suggestion: BP-A2M only needs the $2f+1$ responses from the correct servers to make progress.

(2) f Byzantine servers can not keep requests proposed by a correct server from being chosen by revoking the server: $f + \alpha$ servers are required to revoke a server.

(3) As a proposer, a Byzantine server can not result in gaps in the consensus sequence by not skipping its turn when required: when a correct server has suggested a value, all other servers must either skip its unused turns or suggest values to the turns. Failing to do so will result the server being suspected and revoked, and hence the gap being filled. In the worst case scenario, a Byzantine server does not have to respond until it receives the $(f+\alpha-1)^{th}$ P-RELAY messages. Doing so adds one wide-area message delay. After receiving the $(f+\alpha-1)^{th}$ P-RELAY message, a Byzantine server does not have to send SUGGEST or SKIP messages for its unused turns to all servers: it can omit the messages to up to $f + \alpha - 1$ correct servers without being revoked. The other correct servers, however, relay messages to ensure that the extra latency is at most one wide-area message delay. Thus, the total extra delay a Byzantine server can induce is two wide-area message delays.

5. Related work

We have already explained Paxos [13], Mencius [15], A2M [6] and PBFT [5]. FaB [16] reduces PBFT's proposing phase from two rounds to one round. The improved latency, however, comes at the cost of more replicas: it requires $5f + 1$ replicas to tolerate f failures.

A common technique to improve the latency for BFT protocols is to use speculative execution. Speculative execution can improve the latency for both PBFT and FaB by one round. Zyzzyva [12] is PBFT with client speculation. Introducing the client in the loop not only provides higher throughput but also reduces the latency observed by the clients. Zyzzyva, however, requires the client to be able to talk to all the servers, which can be impractical for multi-site systems, for example, because of inter-site security con-

cerns. A common disadvantage of speculative execution is that a carefully crafted faulty client or server can dramatically reduce the performance of such protocols by forcing an expensive recovery path [7].

Aardvark [7] is the first work to design a robust BFT protocol to provide usable throughput not only during failure-free execution but also during uncivil runs. It accomplishes this by routinely changing the leader that fails to maintain adequate throughput.

Steward [3] is the only BFT protocol we are aware to be designed specifically for multi-site systems. It is a hybrid protocol that replicates servers within a site to provide the illusion of crash-failure semantic for wide-area networks. It is, however, vulnerable when a site is corrupted or behaving selfishly.

Finally, RAM focus on masking faulty servers. Tolerating faulty client behavior is an orthogonal problem and can be dealt with using published techniques [9, 14].

6. Final remarks

RAM is a low-latency replicated state machine protocol designed for wide-area networks consisting of mutually suspicious domains. Unlike approaches such as the one PNUTS [8] uses, RAM does not sacrifice serializability to reduce latency. It is an open question whether RAM's reduced latency is low enough latency to meet the demands of systems like PNUTS.

RAM is built upon three main concepts: MSD, rotating leader, and A2M. MSD is a model of uncivil behavior that assumes trust between clients and server in a site to reduce the latency of local tasks such as returning a value to a request and reading values from the state machine. Although local reads enable lower latency, it implements serializability, but not linearizability. Given that many systems currently sacrifice serializability for performance (*e.g.*, Dynamo [10], PNUTS [8]) this is acceptable for many real systems.

The rotating leader scheme has been used with our protocol Mencius and proved to be efficient. With Byzantine failures, however, tasks such as revocation present subtleties, such as requiring multiple processes to suspect another to guarantee a correct behavior. Although we have discussed and presented a framework for the implementation of such mechanisms we are yet to assess all practical issues that arise with the assumption of Byzantine failures.

Finally, assuming A2M simplifies the protocol by enabling identical Byzantine failures. Abstractly, it is a simple component that enables fewer wide-area messages in our protocol. In practice, it is another component that we have to implement and harden. So far we have no reason to suspect that it is a difficult engineering task to build such a device, but it is part of ongoing work to determine a good way of implementing it. Implementing A2M also requires the use of digital signatures, which are more expensive than the Message Authentication Code (MAC) used by PBFT; this can result in a potential throughput bottleneck. If higher throughput is desired, it can be obtained at the cost of one extra wide-area delay by broadcasting and relaying authenticated PROPOSE and SKIP messages [5, 17]. The use of A2M reflects our engineering tradeoff in favor of lower latency over higher throughput.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 0546686. We would like to thank the anonymous reviewers for their helpful and insightful comments.

References

[1] A. S. Aiyer, L. Alvisi, A. Clement, et al. BAR fault tolerance for cooperative services. *SIGOPS Oper. Syst. Rev.*, 39(5):45–58, 2005.

[2] Y. Amir, B. Coan, J. Kirsch, et al. Byzantine replication under attack. In *DSN 2008*, pages 197–206, Anchorage, AK, USA, 2008.

[3] Y. Amir, C. Danilov, J. Kirsch, et al. Scaling Byzantine fault-tolerant replication to wide area networks. In *DSN 2006*, pages 105–114, Washington, DC, USA, 2006.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[6] B. Chun, P. Maniatis, S. Shenker, et al. Attested append-only memory: making adversaries stick to their word. In *SOSP 2007*, pages 189–204, 2007.

[7] A. Clement, E. Wong, L. Alvisi, et al. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI 2009*, pages 153–168, Boston, MA, USA, 2009.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, et al. PNUTS: Yahoo!'s hosted data serving platform. *VLDB*, 1(2):1277–1288, 2008.

[9] J. Cowling, D. Myers, B. Liskov, et al. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *OSDI 2006*, pages 177–190, Berkeley, CA, USA, 2006.

[10] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[12] R. Kotla, L. Alvisi, M. Dahlin, et al. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP 2007*, pages 45–58, 2007.

[13] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[14] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *ICDCS 2006*, pages 34–43, Lisbon, Portugal, Jul 2006.

[15] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI 2008*, pages 369–384, San Diego, CA, USA, 2008.

[16] J. Martin and L. Alvisi. Fast Byzantine consensus. In *DSN 2005*, pages 402–411, Los Alamitos, CA, USA, 2005.

[17] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.