Mencius: Building Efficient Replicated State Machines for WANs

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

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

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

Abstract

We present a protocol for general state machine replication – a method that provides strong consistency – that has high performance in a wide-area network. In particular, our protocol *Mencius* has high throughput under high client load and low latency under low client load even under changing wide-area network environment and client load. We develop our protocol as a derivation from the well-known protocol Paxos. Such a development can be changed or further refined to take advantage of specific network or application requirements.

1 Introduction

The most general approach for providing a highly available service is to use a replicated state machine architecture [33]. Assuming a deterministic service, the state and function is replicated across a set of servers, and an unbounded sequence of consensus instances is used to agree upon the commands they execute. This approach provides strong consistency guarantees, and so is broadly applicable. Advances in efficient consensus protocols have made this approach practical as well for a wide set of system architectures, from its original application of embedded systems [34] to asynchronous systems. Recent examples of services that use replicated state machines include Chubby [6, 8], ZooKeeper [37] and Boxwood [28].

With the rapid growth of wide-area services such as web services, grid services, and service-oriented architectures, a basic research question is how to provide efficient state machine replication in the wide area. One could choose an application – for example, atomic commit in a service-oriented architecture, and provide an efficient solution for that application (for a large client base and high throughput). Instead, we seek a general solution that only assumes the servers and the clients are spread across a wide-area network. We seek high performance: both high throughput under high client load and low latency under low client load in the face of changing wide-area network environment and client load. And, we seek a solution that comes with a derivation, like the popular consensus protocol Paxos has [22], so it can be modified to apply it to a specific application [15].

Existing protocols such as Paxos, Fast Paxos [25], and CoReFP [13] are not, in general, the best consensus protocols for wide-area applications. For example, Paxos relies on a single leader to choose the request sequence. Due to its simplicity it has high throughput, and requests generated by clients in the same site as the leader enjoy low latency, but clients in other sites have higher latency. In addition, the leader in Paxos is a bottleneck that limits throughput. Having a single leader also leads to an unbalanced communication pattern that limits the utilization of bandwidth available in all of the network links connecting the servers. Fast Paxos and CoReFP, on the other hand do not rely on a single leader. They have low latency under low load, but have lower throughput under high load due to their higher message complexity.

This paper presents *Mencius*[1], a multi-leader state machine replication protocol that derives from Paxos. It is designed to achieve high throughput under high client load and low latency under low client load, and to adapt to changing network and client environments.

The basic approach of Mencius is to partition the sequence of consensus protocol instances among the servers. For example, in a system with three servers, one could assign to server 0 the consensus instances 0, 3, 6 etc, server 1 the consensus instances 1, 4, 7, etc and server 2 the consensus instances 2, 5, 8 etc. Doing this amortizes the load of being a leader, which increases throughput when the system is CPU-bound. When the network is the bottleneck, a partitioned leader scheme more fully utilizes the available bandwidth to increase throughput. It also reduces latency, because clients can use a local server as the leader for their requests; because of the design of Mencius, a client will typically not have to wait for its server to get its turn.

The idea of partitioning sequence numbers among multiple leaders is not original: indeed, it is at the core of a recent patent [26], for the purpose of amortizing server load. To the best of our knowledge, however, Mencius is novel: not only are sequence numbers partitioned, key performance problems such as adapting to changing client load and to asymmetric network bandwidth are addressed. Mencius accomplishes this by building on a simplified version of consensus that we call *simple consensus*. Simple consensus allows servers with low client load to skip their turns without having to have a majority of the servers agree on it first. By opportunistically piggybacking SKIP messages on other messages, Mencius allows servers to skip turns with little or no communication and computation overhead. This allows Mencius to adapt inexpensively to client and network load variance.

The remainder of the paper is as follows. Section 2 describes the wide-area system architecture for which Mencius is designed. Paxos and its performance problems under our system architecture is described in Section 3. Section 4 refines Paxos into Mencius. Section 5 discusses a flexible commit mechanism that reduces latency. Section 6 discusses how to choose parameters. Section 7 evaluates Mencius, Section 8 summarizes related work, and Section 9 discusses future work and open issues. Section 10 concludes the paper.

2 Wide-area replicated state machines

We model a system as n sites interconnected by a wide-area network. Each site has a server and a group of clients. These run on separate processors and communicate through a local-area network. The wide-area network has higher latency and less bandwidth than the local-area networks, and the latency can have high variance. We model the wide-area network as a set of links pairwise connecting the servers. The bandwidth between pairs of servers can be asymmetric and variable.

We do not explicitly consider any dependent behavior of these links. For example, we do not consider issues such as routers that are bottlenecks for communication among three or more sites. This assumption holds when sites are hosted by data centers and links between centers are dedicated. As it turns out, our protocol is quite adaptable to different link behaviors.

Servers communicate with each other through the wide-area network to implement a replicated state machine with 1-copy serializability consistency. Servers can fail by crashing, and perhaps later recovering. The system is asynchronous, in that servers and communication do not need to be timely. Clients access the service by sending requests to their local server via local-area communication. We assume it is acceptable for clients not to make progress while their server is crashed. We discuss relaxing this assumption in Section 9.

Consensus is a fundamental coordination problem that requires a group of processes to agree on a common output, based on their (possibly conflicting) inputs. To implement the replicated state machine, the servers run an unbounded sequence of concurrent instances of consensus [33]. Upon receiving a request from a local client, a server assigns the request (*proposes* a value) using one of the unused consensus instances. Multiple servers may propose different values to the same instance of consensus. All *correct* servers (servers that do not crash) eventually agree on a unique request for each used instance, and this request must have been proposed. When servers agree upon a request for a consensus instance, we say that this request has been *chosen*. Note that choosing a request does not imply that the servers know the outcome of the consensus instance. A server *commits* a request once it *learns* the outcome of the consensus instance. Upon commit, the server requests the application service process to execute the request. If the server is the one that originated the request, then it sends the result back to the client. In addition, a server commits an instance only when it has learned and committed all previous consensus instances. [2]

It is straightforward to see that all correct servers eventually learn and execute the same sequence of requests. If the servers do not skip instances when proposing requests, this sequence also contains no gaps. Thus, if all servers start from the same initial state and the service is deterministic, then the service state will always be consistent across servers and servers will always generate consistent responses.

An efficient implementation of replicated state machines should have high throughput under high client load and low latency under low client load.

For throughput, there are two possible bottlenecks in this service, depending upon the average request size:

Wide-area channels When the average request size is large enough, channels saturate before the servers reach their CPU limit. Therefore, the throughput is determined by how efficiently the protocol is able to propagate requests from its originator to the remaining sites. In this case, we say the system is *network-bound*.

Server processing power When the average request size is small enough, the servers reach their CPU limit first. Therefore, the throughput is determined by the processing efficiency at the bottleneck server. In this case, we say the system is *CPU-bound*.

As a rule of thumb, lower message complexity leads to higher throughput because more network bandwidth is available to send actual state machine commands, and less messages per request are processed.

Servers exchange messages to choose and learn the consensus outcome. Each exchange constitutes a *communication step*. To achieve low latency, it is important to have short chains of wide-area communication steps for the servers to learn the outcome. However, the number of communication steps may not be the only factor impacts latency: high variance on the delivery of message in wide-area networks is also a major contributor [18].

3 Why not Paxos?

Paxos [21, 22] is an efficient asynchronous consensus protocol for replicated state machines. Paxos is a leader-based protocol: one of the servers acts differently than the others, and coordinates the consensus instance. There can be more than one leader at the same time, but during such periods the protocol may not make progress.

Figure 1 illustrates the message flow in a run of a sequence of Paxos instances. Although we show the instances executing sequentially, in practice they can overlap. Each instance of Paxos consists of one or more rounds, and each round can have three phases. Phase 1 (explained in the next paragraph) is only run when there is a leader change. Phase 1 can be simultaneously run for an unbounded number of future instances, which amortizes its cost across all instances that successfully choose a command. Assuming no failures, each server forwards its requests to the leader, which proposes commands (Instance 1 in Figure 1). When the leader receives a proposal, it starts Phase 2 by sending PROPOSE messages (Instance 0 and 1 in Figure 1) that ask the servers (*acceptors* in Paxos terminology) to accept the value. If there are no other leaders concurrently proposing requests, then the servers acknowledge the request with ACCEPT messages. Once the leader receives ACCEPT messages from a majority of the servers, it learns that the value has been chosen and broadcasts a Phase 3 LEARN message to inform the other servers of the consensus outcome. Phase 3 can be omitted by broadcasting ACCEPT messages, which reduces the learning latency for non-leader servers. This option, however, increases the number of messages significantly and so lowers throughput.

When a leader crashes (Instance 2 in Figure 1), the crash is eventually suspected, and another server eventually arises as the new leader. The new leader then starts a higher numbered round and polls the other servers to determine possible commands to propose by running Phase 1 of the protocol. It does this by sending out PREPARE messages and collecting ACK messages from a majority of the servers. Upon finishing Phase 1, the new leader starts Phase 2 to finish any Paxos instances that have been started but not finished by the old leader before crashing.

There are other variants of this protocol, such as Fast Paxos [25] and CoReFP [13], designed to achieve lower latency. Paxos, however, is in general a better candidate for multi-site systems than Fast Paxos and CoReFP because of its simplicity and lower wide-area message complexity, consequently achieving higher throughput.

In the remainder of the paper, we hence compare performance relative only to Paxos. Paxos, however, is still not ideal for wide-area systems:

Unbalanced communication pattern With Paxos, the leader generates and consumes more traffic than the other servers. Figure 1, shows that there is network traffic from replicas to the leader, but no traffic between non-leader replicas. Thus, in a system where sites are pairwise connected, Paxos uses only the channels incident upon the leader, which reduces its ability to sustain high throughput. In addition, during periods of synchrony, only the FWD and PROPOSE messages in Paxos carry significant payload. When the system is network-bound, the volume of these two messages determines the system throughput. In Paxos, FWD is sent from the originator to the leader and PROPOSE is broadcast by the leader. Under high load, the outgoing bandwidth of the leader is a bottleneck, whereas the channels between the non-leaders idle. In contrast, Mencius uses a rotating leader scheme. This not only eliminates the need to send FWD messages, but also gives a more balanced communication pattern, which better utilizes available bandwidth.

Computational bottleneck at the leader The leader in Paxos is a potential bottleneck because it processes more messages than other replicas. When CPU-bound, a system running Paxos reaches its peak capacity when the leader is at full CPU utilization. As the leader requires more processing power than the other servers, the CPU utilization on non-leader servers do not reach their maximum capacity, thus underutilizing the overall processing capacity of the system. The number of messages a leader needs to process for every request grows linearly with the number of servers n, but it remains constant for other replicas. This seriously impacts the scalability of Paxos for larger n. By rotating the leader in Mencius, no single server is a potential bottleneck when the workload is evenly distributed across the sites of the system.

Higher learning latency for non-leader servers While the leader always learns and commits any value it proposes in two communication steps, any other server needs two more communication steps to learn and commit the value it proposes due to the FWD and LEARN messages. With a rotating leader scheme, any server can propose values as a leader. By skipping turns opportunistically when a server has no value to propose, one can achieve the optimal commit delay of two communication steps for any server when there are no concurrent proposals [23]. Concurrent proposals can result in additional delay to commit, but such delays do not always occur. When they do, one can take advantage of commutable operations by having servers execute commands possibly in different, but equivalent orders [24].

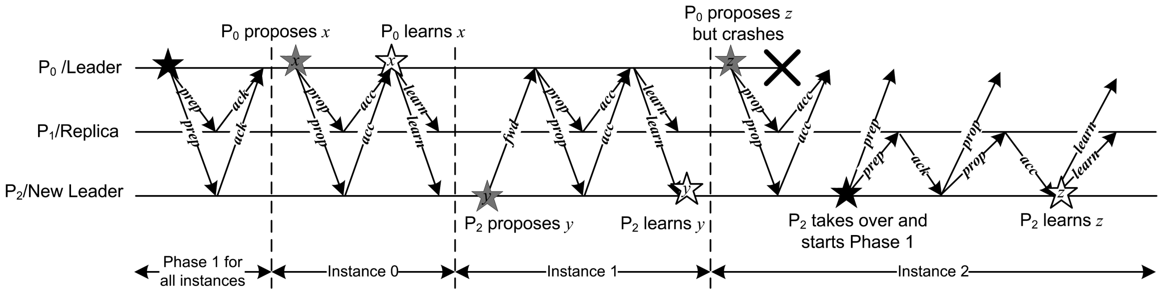


Figure 1: A space time diagram showing the message flow of a sequence of Paxos instances.

4 Deriving Mencius

In this section, we first explain our assumptions and design decisions. We then introduce the concept of simple consensus and use Coordinated Paxos to implement a simple replicated state machine protocol. Finally, we optimize the initial protocol to derive a more efficient one. This last protocol is the one that we call Mencius.

This development of Mencius has two benefits. First, by deriving Mencius from Paxos, Coordinated Paxos, and a set of optimizations and accelerators, it is easier to see that Mencius is correct. Second, one can continue to refine Mencius or even derive a new version of Mencius to adapt it to a particular application.

4.1 Assumptions

We make the following assumptions about the system. We omit a formal description of the assumptions, and we refer readers to [9, 10, 21, 22] for details.

Crash process failure Like Paxos, Mencius assumes that servers fail by crashing and can later recover. Servers have access to stable storage, which they use to recover their states prior to failures.

Unreliable failure detector Consensus is not solvable in an asynchronous environment when even a single process can fail [14]. Like many other asynchronous consensus protocols, Mencius utilizes a failure detector oracle to circumvent the impossibility result. Like Paxos, it relies on the failure detector only for liveness – Mencius is safe even when the failure detector makes an unbounded number of mistakes. Mencius requires that, eventually, all faulty servers and only faulty servers are suspected by the failure detector. In practice, such a failure detector can be implemented by increasing timeouts exponentially. A discussion on our requirements on failure detector can be found in [29].

Asynchronous FIFO communication channel Since we use TCP as the underlying transport protocol, we assume FIFO channels and that messages between two correct servers are eventually delivered. This is a strictly stronger assumption compared to the one of Paxos. Had we instead decided to use UDP, we would have to implement our own message retransmission and flow control at the application layer. Assuming FIFO enables optimizations discussed in Section 4.4. These optimizations, however, are applicable only if both parties of a channel are available and a TCP connection is established. When servers fail and recover after long periods, implementing FIFO channels is impractical as it may require buffering a large number of messages. Mencius uses a separate recovery mechanism that does not depend on FIFO channels (see Section 4.5).

4.2 Simple consensus and Coordinated Paxos

As explained in Section 3, Paxos only allows the leader to propose values. We instead have servers take turns in proposing values. By doing so, servers do not contend when proposing values if there are no failures and no false suspicions. We take advantage of this fact with *simple consensus*.

Simple consensus is consensus in which the values a server can propose are restricted. Let *no-op* be a state machine command that leaves the state unchanged and that generates no response. In simple consensus, only one special server, which we call the *coordinator*, can propose any command (including *no-op*); the others can only propose *no-op*.[3] With Mencius, a replicated state machine runs concurrent instances of simple consensus.

For each instance, one server is designated as the coordinator. Also, the assignment scheme of instances to coordinators is known by all servers. To guarantee that every server has a turn to propose a value, we require that: (1) every server is the coordinator of an unbounded number of instances, and (2) for every server p there is a bounded number of instances assigned to other servers between consecutive instances that p coordinates. A simple scheme assigns instance $cn + p$ to server p, where $c \in \mathbb{N}_0$ and $p \in \{0, \ldots, n-1\}$. Without loss of generality, we assume this scheme for the rest of this paper.

A benefit of using simple consensus is that servers can learn a skipped *no-op* without having to have a majority of servers to agree on it first. As a result, SKIP messages have the minimal learning latency of just one one-way message delay. This ability combined with two optimizations discussed in Section 4.4 makes it possible for the servers to propose *no-op* at very little cost of both com-

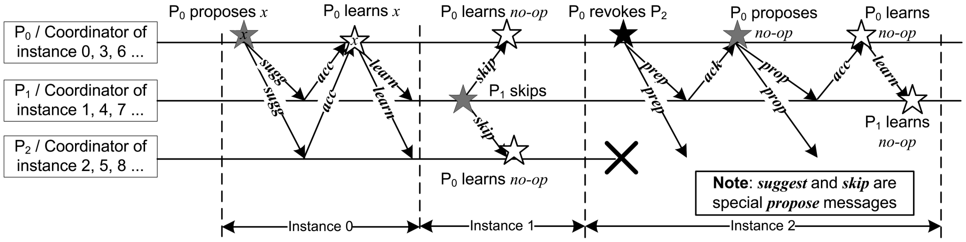


Figure 2: The message flow of suggest, skip and revoke in Coordinated Paxos.

munication and computation overhead. This gives Mencius the ability to adapt quickly and cheaply to changing client load and network bandwidth. Another benefit of simple consensus is discussed in Section 5: by restricting the values a non-coordinator can propose, one can implement a flexible commit mechanism that further reduces Mencius's latency.

Since simple consensus only restricts the initial value a server can propose, any implementation of consensus, including Paxos, can be used to solve simple consensus.

We use, however, an efficient variant of Paxos to implement simple consensus. We call it *Coordinated Paxos* (see [29] for the proof of correctness and the pseudo code). In each instance of Coordinated Paxos, all servers agree that the coordinator is the default leader, and start from the state in which the coordinator had run Phase 1 for some initial round r. Such a state consists of a promise not to accept any value for any round smaller than r. A server can subsequently initiate the following actions, as shown in Figure 2:

Suggest The coordinator *suggests* a request v by sending PROPOSE messages with payload v in round r (Instance 0 in Figure 2). We call these PROPOSE messages SUGGEST messages.

Skip The coordinator *skips* its turn by sending PROPOSE messages that proposes *no-op* in round r (Instance 1 in Figure 2). We call these PROPOSE messages SKIP messages. Note that because all other servers can only propose *no-op*, when the coordinator proposes *no-op*, any server learns that *no-op* has been chosen as soon as it receives a SKIP message from the coordinator.

Revoke When suspecting that the coordinator has failed, some server will eventually arise as the new leader and revoke the right of the coordinator to propose a value. The new leader does so by trying to finish the simple consensus instance on behalf of the coordinator (Instance 2 in Figure 2). Just like a new Paxos leader would do, it starts Phase 1 for some round $r' > r$. If Phase 1 indicates no value may have been chosen, then the new leader proposes *no-op* in Phase 2. Otherwise, it proposes the possible consensus outcome indicated by Phase 1.

The actions *suggest*, *skip* and *revoke* specialize mechanisms that already exist in Paxos. Making them explicit, however, enables more efficient implementations in wide-area networks. The main differences between Coordinated Paxos and Paxos are the following: (1) Coordinated Paxos starts from a different (but safe) state; and (2) a server learns *no-op* upon receiving a SKIP message from the coordinator, and can act accordingly.

4.3 A simple state machine

We now construct an intermediate protocol $\mathcal{P}$ that implements replicated state machines. At high level, $\mathcal{P}$ runs an unbounded sequence of simple consensus instances and each instance is solved with Coordinated Paxos. We describe $\mathcal{P}$ using four rules that determine the behavior of a server and argue that $\mathcal{P}$ is correct using these rules. The pseudo code of $\mathcal{P}$ is in [29]. In Section 4.4, we derive Mencius from $\mathcal{P}$.

While consistency (safety) is guaranteed by Coordinated Paxos, $\mathcal{P}$ still needs to handle duplicate requests that arise from clients submitting requests multiple times due to timeouts. This can be done by using any well-known technique, such as assuming idempotent requests or by recording committed requests and checking for duplicates before committing. We assume such a technique is used.

For liveness, we use Rule 1-4 to ensure that any client request sent to a correct server eventually commits. To minimize the delay in learning, a server suggests a value immediately upon receiving it from a client.

Rule 1 Each server p maintains its next simple consensus sequence number I_p. We call I_p the *index* of server p. Upon receiving a request from a client, a server p suggests the request to the simple consensus instance I_p and updates I_p to the next instance it will coordinate.

Rule 1 by itself performs well only when all servers suggest values at about the same rate. Otherwise, the index of a server generating requests more rapidly will increase faster than the index of a slower server. Servers cannot commit requests before all previous requests are committed, and so Rule 1 commits requests at the rate of the slowest server. In the extreme case that a server suggests no request for a long period of time, the state machine stalls, preventing a potentially unbounded number of requests from committing. Rule 2 uses a technique similar to logical clocks [20] to overcome this problem.

Rule 2 If server p receives a SUGGEST message for instance i and $i > I_p$, before accepting the value and sending back an ACCEPT message, p updates I_p such that its new index $I'_p = min\{k : p$ coordinates instance $k \wedge k > i\}$. p also executes skip actions for each of the instances in range $[I_p, I'_p)$ that p coordinates.

With Rule 2, slow servers skip their turns. Consequently, the requests that fast servers suggest do not have to wait for slow servers to have requests to suggest before committing. However, a crashed server does not broadcast SKIP messages, and such a server can prevent others from committing. Rule 3 overcomes this problem.

Rule 3 Let q be a server that another server p suspects has failed, and let C_q be the smallest instance that is coordinated by q and not learned by p. p revokes q for all instances in the range $[C_q, I_p]$ that q coordinates. [4]

If any correct server p suggests a value v to instance i, a server updates its index to a value larger than i upon receiving this SUGGEST message. Thus, according to Rule 2, every correct server r eventually proposes a value (either by skipping or by suggesting) to every instance smaller than i that r coordinates, and all non-faulty servers eventually learn the outcome of those instances. For instances that faulty servers coordinate, according to Rule 3, non-faulty servers eventually revoke them, and non-faulty servers eventually learn the outcome. Thus, all instances prior to i are eventually learned, and request v eventually commits, assuming that p is not falsely suspected by other servers.

False suspicions, however, are possible with unreliable failure detectors. We add Rule 4 to allow a server to suggest a request multiple times upon false suspicions.

Rule 4 If server p suggests a value $v \neq$ *no-op* to instance i, and p learns that *no-op* is chosen, then p suggests v again.

As long as p is not permanently falsely suspected, p will continue to re-suggest v, and v will be eventually chosen. In practice, a period of no false suspicions only needs to hold long enough for p to re-suggest v and have it chosen for the protocol to make progress.

Due to space constraints, we omit the proof of correctness for $\mathcal{P}$ and refer interested readers to [29].

4.4 Optimizations

Protocol $\mathcal{P}$ is correct but not necessarily efficient. It always achieves the minimal two communication steps for a proposing server to learn the consensus value, but its message complexity varies depending on the rates at which the servers suggest values. The worst case is when only one server suggests values, in which case the message complexity is $(n-1)(n+2)$ due to the broadcast SKIP messages that Rule 2 generates.

Consider the case where server p receives a SUGGEST message for instance i from server q. As a result, p skips all of its unused instances smaller than i (Rule 2). Let the first instance that p skips be i_1 and the last instance p skips be i_2. Since p needs to acknowledge the SUGGEST message of q with an ACCEPT message, p can piggyback the SKIP messages on the ACCEPT message. Since channels are FIFO, by the time q receives this ACCEPT message, q has received all the SUGGEST messages p sent to q before sending the ACCEPT message to q. This means that p does not need to include i_1 in the ACCEPT message: i_1 is the first instance coordinated by p that q does not know about. Similarly, i_2 does not need to be included in the ACCEPT message because i_2 is the largest instance smaller than i and coordinated by p. Since both i and p are already included in the ACCEPT message, there is no need for any additional information: all we need to do is augmenting the semantics of the ACCEPT message. In addition to acknowledging the value suggested by q, this message now implies a promise from p that it will not suggest any client requests to any instances smaller than i in the future. This gives us the first optimization:

Optimization 1 p does not send a separate SKIP message to q. Instead, p uses the ACCEPT message that replies the SUGGEST to promise not to suggest any client requests to instances smaller than i in the future.

Protocol $\mathcal{P}$ with Optimization 1 implements replicated state machines correctly.

We can also apply the same technique to the SKIP messages from p to other servers. Instead of using ACCEPT messages, we piggyback the SKIP messages on future SUGGEST messages from p to another server r:

Optimization 2 p does not send a SKIP message to r immediately. Instead, p waits for a future SUGGEST message from p to r to indicate that p has promised not to suggest any client requests to instances smaller than i.

Note that Optimization 2 can potentially defer the propagation of SKIP messages from p to r for an unbounded period of time. For example, consider three servers p_0, p_1, p_2. Only p_0 suggests values for instance 0, 3, 6, and so on. p_0 always learns the result for all instances by means of the ACCEPT messages from p_1 and p_2. Server p_1, however, learns all values that p_0 proposes, and it knows which instances it is skipping, but it does not learn that p_2 skips, such as for instance 2 in this example. This leaves gaps in the view of p_1 of the consensus sequence and prevents p_1 from committing values learned in instance 3, 6, and so on. Similarly, p_2 does not learn that p_1 is skipping and prevents p_2 from committing values learned in 3, 6, and so on.

This problem only occurs between two idle servers p_1 and p_2: any value suggested by either server will propagate the SKIP messages in both directions and hence fill in the gaps. Fortunately, while idle, neither p_1 nor p_2 is responsible for generating replies to the clients. This means that, from the client perspective, its individual requests are still being processed in a timely manner, even if p_1 and p_2 are stalled. We use a simple accelerator rule to limit the number of outstanding SKIP messages before p_1 and p_2 start to catch up:

Accelerator 1 A server p propagates SKIP messages to r if the total number of outstanding SKIP messages to r is larger than some constant α, or the messages have been deferred for more than some time τ.

Note that Optimization 2 and Accelerator 1 can only delay the propagation of SKIP messages for a bounded amount of time. Since $\mathcal{P}$ only relies on the eventual delivery of messages for liveness, adding Optimization 2 and Accelerator 1 to protocol $\mathcal{P}$ still implements replicated state machines correctly.

Given that the number of extra SKIP messages generated by Accelerator 1 are negligible over the long run, the amortized wide-area message complexity for Mencius is $3n-3$ ($(n-1)$ SUGGEST, ACCEPT and LEARN messages each), the same as Paxos when FWD is not considered.

We can also reduce the extra cost generated by the revocation mechanism. If server q crashes, revocations need to be issued for every simple consensus instance that q coordinates. By doing this, we increase both the latency and message complexity due to the use of the full three phases of Paxos. A simple idea is to revoke all q's future turns, which irreversibly chooses *no-op* for all q's further turns. However, q may need to suggest values in the future, either because q was falsely suspected or because it recovers. A better idea is the following:

Optimization 3 Let q be a server that another server p suspects has failed, and let C_q be the smallest instance that is coordinated by q and not learned by p. For some constant β, p revokes q for all instances in the range $[C_q, I_p + 2\beta]$ that q coordinates if $C_q < I_p + \beta$.

Optimization 3 allows p to revoke q at least β instances in advance before p suggests a value to some instance i greater than C_q. By tuning β, we ensure that by the time p learns the outcome of instance i, all instances prior to i and coordinated by q are revoked and learned. Thus, p can commit instance i without further delay. Since Optimization 3 also requires revocations being issued in large blocks, the amortized message cost is small.

Note that Optimization 3 can only exclude the actions of a falsely suspected server for a bounded number of instances. Since we assume the failure detector will eventually be accurate, such false suspicions will eventually cease. So, using Optimization 3 does not affect the liveness of the protocol.

Optimization 3 addresses the common case where there are no false suspicions. When a false suspicion does occur, it may result in poor performance while servers are falsely suspected. We consider the poor performance in this case acceptable because we assume false suspicions occur rarely in practice and the cost of recovery from a false suspicion is small (see Section 6).

Mencius is $\mathcal{P}$ combined with Optimizations 1-3 and Accelerator 1. From the above arguments, Mencius implements replicated state machines correctly. Due to lack of space, we omit the proof of correctness and the pseudo code, both of which can be found in [29].

Mencius, being derived from Paxos, has the same quorum size of $f+1$. This means that up to f servers can fail among a set of $2f + 1$ servers. Paxos incurs temporarily reduced performance when the leader fails. Since all servers in Mencius act as a leader for an unbounded number of instances, Mencius has this reduced performance when *any* server fails. Thus, Mencius has higher performance than Paxos in the failure-free case at the cost of potentially higher latency upon failures. Note that higher latency upon failures also depends on other factors such as the stability of the communication network.

4.5 Recovery

In this section, we outline how Mencius recovers from failures. Due to lack of space, we omit the details.

Temporary broken TCP connection We add an application layer sequence number to Mencius's messages. FIFO channels are maintained by retransmitting missing messages upon reestablishing the TCP connection.

Short term failure Like Paxos, Mencius logs its state to stable storage and recovers from short term failures by replaying the logs and learning recent chosen requests from other servers.

Long term failure It is impractical for a server to recover from a long period of down time by simply learning missing sequences from other servers, since this requires correct servers to maintain an unbounded long log. The best way to handle this, such as with checkpoints or state transfer [8, 27], is usually application specific.

5 Commit delay and out-of-order commit

In Paxos, the leader serializes the requests from all the servers. For purposes of comparison, assume that Paxos is implemented, like Mencius, using FIFO channels. If the leader does not crash, then each server learns the requests in order, and can commit a request as soon as it learns the request. The leader can commit a request as soon as it collects ACCEPT messages from a quorum of $f+1$ servers, and any other server will have an additional round trip delay due to the FWD and LEARN messages.

While a Mencius server can commit the request in just one round trip delay when there is no contention, commits may have to be delayed up to two communication steps when there are concurrent suggestions.

For example, in the scenario illustrated in Figure 3, server p_0 suggests x to instance 0 concurrently with p_1 suggesting y to instance 1. p_1 receives the SUGGEST

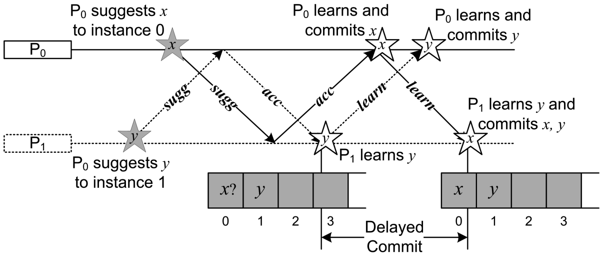

P$_0$ suggests x to instance 0
P$_0$ learns and commits x
P$_0$ learns and commits y
P$_0$
P$_1$
sugg
sugg
acc
acc
learn
learn
P$_1$ learns y and commits x, y
P$_0$ suggests y to instance 1
P$_1$ learns y
x?
y
0
1
2
3
Delayed Commit
x
y
0
1
2
3

Figure 3: Delayed commit.

message for x from p_0 before it receives the ACCEPT message for y. Upon receiving the ACCEPT for y from p_0, p_1 learns that y has been chosen for instance 1, but cannot commit y yet as it has only accepted but not learned x for instance 0. In this case, p_1 cannot commit y until it receives the LEARN message for x from p_0, at which point it can commit both x and y at once. We say that y experiences a *delayed commit* at p_1.

The delay can be up to two communication steps, since p_1 must learn y in between accepting x and learning x for a delayed commit to occur. If p_1 learns y after it learns x, then there is clearly no extra delay. If p_1 learns y before it accepts x, then p_0 must have accepted y before suggesting x because of the FIFO property of the channel. In this case, according to Rule 2, p_0 must have skipped instance 0, which contradicts the assumption that p_0 suggested x to instance 0. Thus, the extra delay to commit y can be as long as one round trip communication between p_0 and p_1 (p_1 sends ACCEPT to p_0 and p_0 sends LEARN back), *i.e.*, up to two communication steps. We can reduce the upper bound of delayed commit to one communication step by broadcasting ACCEPT messages and eliminating LEARN messages. This reduction gives Mencius an optimal commit delay of three communication steps when there are concurrent proposals [23] at the cost of higher message complexity and thus lower throughput.

Because delayed commit arises with concurrent suggestions, it becomes more of a problem as the number of suggestions grows. In addition, delayed commit impacts the commit latency but not overall throughput: over a long period of time, the total number of requests committed is independent of delayed commits.

Out-of-order commit We can mitigate the effects of delayed commit with a simple and more flexible commit mechanism that allows x and y to be executed in any order when they are *commutable*, *i.e.*, executing x followed by y produces the same system state as executing y followed by x. By the definition of simple consensus, when p_1 receives the SUGGEST message for x, it knows that only x or *no-op* can be chosen for instance 0. Since *no-op* commutes with any request, upon learning y, p_1 can commit y before learning x and send the result back to the client without any delay when x and y are commutable. We call this mechanism *out-of-order commit* and evaluate its effectiveness in Section 7.5. We implement out-of-order commit in Mencius by tracking the dependencies between the requests and by committing a request as soon as all requests it depends on have been committed. This technique can not be applied to Paxos as easily, because Paxos is based on consensus, which does not have any restriction on the value a server can propose – the key for Mencius to guarantee safety while allowing out-of-order commit.

6 Choosing parameters

Accelerator 1 and Optimization 3 use three parameters: α, β and τ. We discuss here strategies for choosing these parameters.

Accelerator 1 limits the number of outstanding SKIP messages between two idle server p_1 and p_2 before they start to catch up. It bounds both the amount of time (τ) and number of outstanding messages (α).

When choosing τ, it should be large enough so that the cost of SKIP messages can be amortized. But, a larger τ adds more delay to the propagation of SKIP messages, and so results in extra commit delay for requests learned at p_1 and p_2. Fortunately, when idle, neither p_1 nor p_2 generates any replies to the clients, and so such extra delay has little impact from a client's point of view. For example, in a system with 50 ms one-way link delay, we can set τ to the one-way delay. This is a good value because: (1) With $\tau = 50$ ms, Accelerator 1 generates at most 20 SKIP messages per second, if α is large enough. The network resource and CPU power needed to transmit and process these messages are negligible; and (2) The extra delay added to the propagation of the SKIP messages is at most 50 ms, which could occur anyway due to network delivery variance or packet loss.

α limits the number of outstanding SKIP messages before p_1 and p_2 start to catch up: if τ is large enough, α SKIP messages are combined into just one SKIP message, reducing the overhead of SKIP messages by a factor of α. For example, we set α to 20 in our implementation, which reduces the cost of SKIP message by 95%.

β defines an interval of instances: if a server q is crashed and I_p is the index of a non-faulty server p, then in steady state all instances coordinated by q and in the range $[I_p, I_p + k]$ for some $k : \beta \leq k \leq 2\beta$ are revoked. Choosing a large β guarantees that while crashed, q's inactivity will not slow down other servers. It, however, makes the indexes of q and other servers more out of synchronization when q recovers from a false suspicion or a failure. Nonetheless, the overhead of having a large β is negligible. Upon recovery, q will learn the instances it coordinates that have been revoked. It then updates its index to the next available slot and suggests the next client request using that instance. Upon receiving the SUGGEST message, other replicas skip their turns and catch up with q's index (Rule 2). The communication overhead of skipping is small, as discussed in Optimization 1 and 2. The computation overhead of skipping multiple consecutive instances at once is also small, since an efficient implementation can easily combine their states and represent them at the cost of just one instance. While setting β too large could introduce problems with consensus instance sequence number wrapping, any practical implementation should have plenty of room to choose an appropriate β.

Here is one way to calculate a lower bound for β. Revocation takes up to two and a half round trip delays. Let i be an instance of server q that is revoked. To avoid delayed commit of some instance $i' > i$ at a server p, one needs to start revoking i two and a half round trips in advance of instance i' being learned by p. In our implementation with a round trip delay of 100 ms and with $n = 3$, the maximum throughput is about $10,000$ operations per second. Two and a half round trip delays are 250 ms, which, at maximum throughput, is $2,500$ operations. All of these operations could be proposed by a single server, and so the instance number may advance by as many as $3 \times 2,500 = 7,500$ in any 250 ms interval. Thus, if $\beta \geq 7,500$, then in steady state no instances will suffer delayed commit arising from q being crashed. Taking network deliver variance into account, we set $\beta = 100,000$, which is a conservative value that is more than ten times the lower bound, but still reasonably small even for the 32-bit sequence number space in our implementation.

7 Evaluation

We ran controlled experiments in the DETER testbed [5] to evaluate the performance of Mencius and Paxos. We used TCP as the transport protocol and implemented both protocols in C++. Here are some implementation details:

API Both Paxos and Mencius implement two simple API calls: PROPOSE(v) and ONCOMMIT(v). An application calls PROPOSE to issue a request, and the state machine upcalls the application via ONCOMMIT when the request is ready to commit. When out-of-order commit is enabled, Mencius uses a third upcall ISCOMMUTE(u, v) to ask the application if two requests are commutable.

Nagle's algorithm Nagle's algorithm [30] is a technique in TCP for improving the efficiency of wide-area communication by batching small messages into larger ones. It does so by delaying sending small messages and waiting for data from the application. In our implementation, we can instruct servers to dynamically turn on or turn off Nagle's algorithm.

Parameters We set the parameters that control Accelerator 1 and Optimization 3 to $\alpha = 20$ messages, $\tau = 50$ ms, and $\beta = 100,000$ instances.

7.1 Experimental settings

To compare the performance of Mencius and Paxos, we use a simple, low-overhead application that enables commutable operations. We chose a simple read/write register service of κ registers. The service implements a read and a write command. Each command consists of the following fields: (1) operation type – read or write (1 bit); (2) register name (2 bytes); (3) the request sequence number (4 bytes); and (4) ρ bytes of dummy payload. All the commands are ordered by the replicated state machine in our implementation. When a server commits a request, it executes the action, sends a zero-byte reply to the client and logs the first three fields along with the client's ID. We use the logs to verify that all servers learn the same client request sequence; or, when reordering is allowed, that the servers learned compatible orders. Upon receiving the reply from the server, the client computes and logs the latency of the request. We use the client-side log to analyze experiment results.

We evaluated the protocols using a three-server clique topology for all but the experiments in Section 7.4. This architecture simulated three data centers (A, B and C) connected by dedicated links. Each site had one server node running the replicated register service, and one client node that generated all the client requests from that site. Each node was a 3.0 GHz Dual-Xeon PC with 2.0 GB memory running Fedora 6. Each client generated requests at either a fixed rate or with inter-request delays chosen randomly from a uniform distribution. The additional payload size ρ was set to be 0 or $4,000$ bytes. 50% of the requests were reads and 50% were writes. The register name was uniformly chosen from the total number of registers the service implemented. A virtual link was set up between each pair of sites using the DummyNet [31] utility. Each link had a one-way delay of 50 ms. We also experimented with other delay settings such as 25 ms and 100 ms, but do not report these results here because we did not observe significant differences in the findings. The link bandwidth values varied from 5 Mbps to 20 Mbps. When the bandwidths were chosen within this range, the system was network-bound when $\rho = 4,000$ and CPU-bound when $\rho = 0$. Except where noted, Nagle's algorithm was enabled.

In this section, we use "Paxos" to denote the register service implemented with Paxos, "Mencius" to denote the register service using Mencius and with out-of-order commit disabled, and "Mencius-κ" to denote the service using Mencius with κ total registers and out-of-order commit enabled (*e.g.*, Mencius-128 corresponds to the service with 128 registers). Given the read/write ratio, requests in Mencius-κ can be moved up, on average,

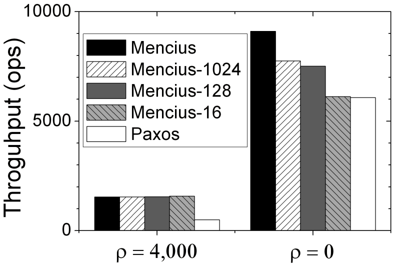


Figure 4: Throughput for 20 Mbps bandwidth

0.75κ slots before reaching an incommutable request. We used κ equal to 16, 128, or $1,024$ registers to represent a service with low, moderate and a high likelihood of adjacent requests being commutable, respectively.

We first describe, in Section 7.2, the throughput of the service both when it is CPU-bound and when it is network-bound, and we show the impact of asymmetric channels and variable bandwidth. In both cases, Mencius has higher throughput. We further evaluate both protocols under failures in Section 7.3. In Section 7.4 we show that Mencius is more scalable than Paxos. In Section 7.5 we measure latency and observe the impact of delayed commit. In general, as load increases, the commit latency of Mencius degrades from being lower than Paxos to being the same as the one of Paxos. Reordering requests decreases the commit latency of Mencius. Finally, we show that the impact of variance in network latency is complex.

7.2 Throughput

To measure throughput, we use a large number of clients generating requests at a high rate. Figure 4 shows the throughput of the protocols, for a fully-connected topology with 20 Mbps available for each link, and a total of 120 Mbps available bandwidth for the whole system.

When $\rho = 4,000$, the system was network-bound: all four Mencius variants had a fixed throughput of about $1,550$ operations per sec (ops). This corresponds to 99.2 Mbps, or 82.7% utilization of the total bandwidth, not counting the TCP/IP and MAC header overhead. Paxos had a throughput of about 540 ops, or one third of Mencius's throughput: Paxos is limited by the leader's outgoing bandwidth.

When $\rho = 0$, the system is CPU-bound. Paxos presents a throughput of $6,000$ ops, with 100% CPU utilization at the leader and 50% at the other servers. Mencius's throughput under the same condition was $9,000$ ops, and all three servers reached 100% CPU utilization. Note that the throughput improvement for Mencius was in proportion to the extra CPU processing power available. Mencius with out-of-order commit enabled had lower throughput compared to Mencius with this feature disabled because Mencius had to do the extra work of dependency tracking. The throughput drops as the total number of registers decreases because with fewer registers there is more contention and dependencies to handle.

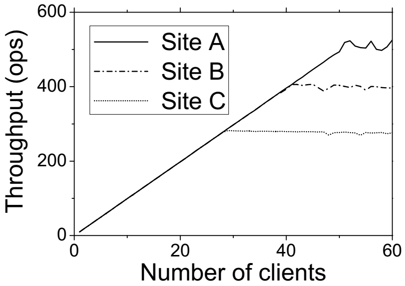


Figure 5: $\rho = 4,000$ with asymmetric bandwidth

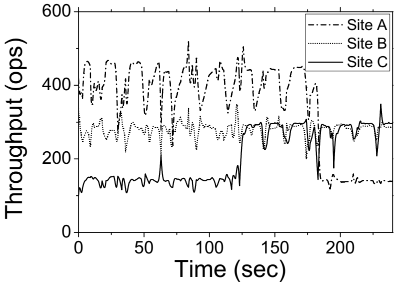


Figure 6: Mencius dynamically adapts to changing network bandwidth ($\rho = 4,000$)

Figure 5 demonstrates Mencius's ability to use available bandwidth even when channels are asymmetric with respect to bandwidth. Here, we set the bandwidth of the links $A \rightarrow B$ and $A \rightarrow C$ to 20 Mbps, links $B \rightarrow C$ and $B \rightarrow A$ to 15 Mbps and links $C \rightarrow B$ and $C \rightarrow A$ to 10 Mbps. We varied the number of clients, ensuring that each site had the same number of clients. Each client generated requests at a constant rate of 100 ops. The additional payload size ρ was $4,000$ bytes. As we increased the number of clients, site C eventually saturated its outgoing links first; and from that point on committed requests at a maximum throughput of 285 ops. In the meanwhile, the throughput at both A and B increased until site B saturated its outgoing links at 420 ops. Finally site A saturated its outgoing links at 530 ops. As expected, the maximum throughput at each site is proportional to the outgoing bandwidth (in fact, the minimum bandwidth).

Figure 6, shows Mencius's ability to adapt to changing network bandwidth. We set the bandwidth of links $A \rightarrow B$ and $A \rightarrow C$ to 15 Mbps, links $B \rightarrow A$ and $B \rightarrow C$ to 10 Mbps, and link $C \rightarrow A$ and $C \rightarrow B$ to 5 Mbps. Each site had a large number of clients generating enough requests to saturate the available bandwidth. Site A, B and C initially committed requests with throughput of about 450 ops, 300 ops, and 150 ops respectively, reflecting the bandwidth available to them. At time $t = 60$ seconds, we dynamically increased the bandwidth of link $C \rightarrow A$ from 5 Mbps to 10 Mbps. With the exception of a spike, C's throughput did not increase because it is limited by the 5 Mbps link from C to B. At $t = 120$ seconds, we dynamically increased the bandwidth of link $C \rightarrow B$ from 5 Mbps to 10 Mbps. This time, site C's throughput doubles accordingly. At $t = 180$ seconds, we dynamically decreased the bandwidth of link $A \rightarrow C$ from 15 Mbps to 5 Mbps. The throughput at site A dropped, as expected, to one third.

In summary, Mencius achieves higher throughput compared to Paxos under both CPU-bound and network-bound workload. Mencius also fully utilizes available bandwidth and adapts to bandwidth changes.

7.3 Throughput under failure

In this section, we show throughput during and after a server failure. We ran both protocols with three servers under network-bound workload ($\rho = 4,000$). After 30 seconds, we crashed one server. We implemented a simple failure detector that suspects a peer when it detects the loss of TCP connection. The suspicion happened quickly, and so we delayed reporting the failure to the suspecting servers for another five seconds. Doing so made it clearer what occurs during the interval when a server's crash has not yet been suspected.

Figure 7(a) shows Mencius's instantaneous throughput observed at server p_0 when we crash server p_1. The throughput is roughly 850 ops in the beginning, and quickly drops to zero when p_1 crashes. During the period the failure remains unreported, both p_0 and p_2 are still able to make progress and learn instances they coordinate, but cannot commit these instances because they have to wait for the consensus outcome of the missing instances coordinated by p_1. When the failure detector reports the failure, p_0 starts revocation against p_1. At the end of the revocation, p_0 and p_2 learn of a large block of *no-op*s for instances coordinated by p_1. This enables p_0 to commit all instances learned during the five second period in which the failure was not reported, which results in a sharp spike of $3,600$ ops. Once these instances are committed, Mencius's throughput stabilizes at roughly 580 ops. This is two thirds of the rate before the failure, because there is a reduction in the available bandwidth (there are fewer outgoing links), but it is still higher than that of Paxos under the same condition.

Figure 7(b) shows Paxos's instantaneous throughput observed at server p_1 when we crash the leader p_0. Throughput is roughly 285 ops before the failure, and it quickly drops to zero when p_0 crashes because the leader serializes all requests. Throughput remains zero for five seconds until p_1 becomes the new leader, which then starts recovering previously unfinished instances. Once it finishes recovering such instances, Paxos's throughput goes back to 285 ops, which was roughly the throughput before the failure of p_0. Note that at $t = 45$ seconds, there is a sharp drop in the throughput observed at p_1. This is due to duplicates: upon discovering the crash of p_0, both p_1 and p_2 need to re-propose requests that have been forwarded to p_0 and are still unlearned. Some of the requests, however, have sequence numbers (assigned by p_0) and have been accepted by either p_1 or p_2. Upon taking leadership, p_1 revokes such instances, hence resulting in duplicates. In addition, the throughput at p_1 has higher variance after the failure than before. This is consistent with our observation that the Paxos leader sees higher variance than other servers.

Figure 7(c) shows Paxos's instantaneous throughput of leader p_0 when p_1 crashes. There is a small transient drop in throughput but since the leader and a majority of servers remain operational, throughput quickly recovers.

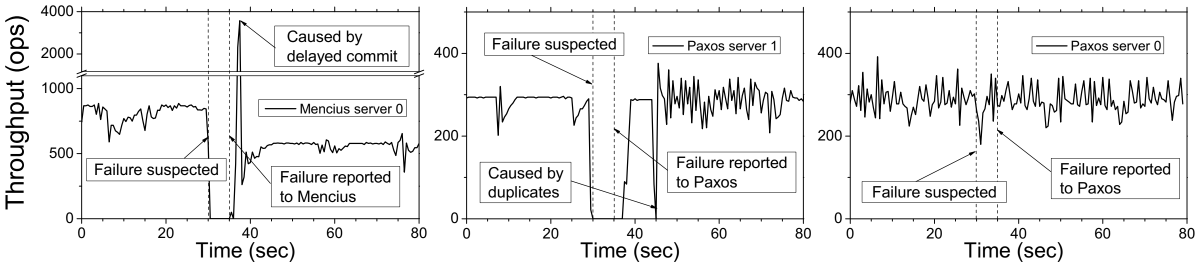
To summarize, Mencius temporarily stalls when any of the servers fails while Paxos temporarily stalls only when the leader fails. Also, the throughput of Mencius drops after a failure because of a reduction on available bandwidth, while the throughput of Paxos does not change since it does not use all available bandwidth.

7.4 Scalability

For both Paxos and Mencius, availability increases by increasing the number of servers. Given that wide-area systems often target an increasing population of users, and sites in a wide-area network can periodically disconnect, scalability is an important property.

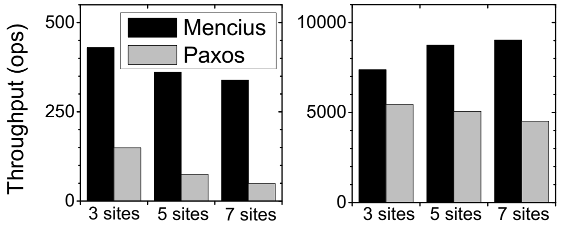
We evaluated the scalability of both protocols by running them with a state machine ensemble of three, five and seven sites. We used a star topology where all sites connected to a central node: these links had a bandwidth of 10 Mbps and 25 ms one-way delay. We chose the star topology to represent the Internet cloud as the central node models the cloud. The 10 Mbps link from a site represents the aggregated bandwidth from that site to all other sites. We chose 10 Mbps because it is large enough to have a CPU-bound system when $\rho = 0$, but small enough so that the system is network-bound when $\rho = 4,000$. When $n = 7$, 10 Mbps for each link gives a maximum demand of 70 Mbps for the central node, which is just under its 100 Mbps capacity. The 25 ms one-way delay to the central node gives an effective 50 ms one-way delay between any two sites. Because we only consider throughput in this section, network latency is irrelevant. To limit the number of machines we use, we chose to run the clients and the server on the same physical machine at each site. Doing this takes away some of the CPU processing power from the server; this is equivalent to running the experiments on slower machines under CPU-bound workload ($\rho = 0$), and has no effect under network-bound workload ($\rho = 4,000$).

When the system is network-bound, increasing the number of sites (n) makes both protocols consume more



(a) One Mencius server crashes (b) Paxos leader crashes (c) One Paxos non-leader crashes

Figure 7: Mencius and Paxos's throughput under failure



(a) $\rho = 4,000$ (b) $\rho = 0$

Figure 8: Throughput vs. number of sites

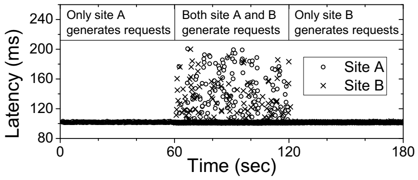


Figure 9: Mencius's commit latency when client load shifts from one site to another

bandwidth per request: each site sends a request to each of the remaining $n-1$ sites. Since Paxos is limited by the leader's total outgoing bandwidth, its throughput is in proportion to $\frac{1}{n-1}$. Mencius, on the other hand, can use the extra bandwidth provided by the new sites, and so the throughput is in proportion to $\frac{n}{n-1}$. Figure 8(a) shows both protocols' throughput with $\rho = 4,000$. Mencius started with a throughput of 430 ops with three sites, approximately three times higher than Paxos's 150 ops under the same condition. When n increased to five, Mencius's throughput drops to 360 ops ($84\% \approx (\frac{5}{4})/(\frac{3}{2})$), while Paxos's drops to 75 ops ($50\% = (\frac{1}{4})/(\frac{1}{2})$). When n increased to seven, Mencius's throughput dropped to 340 ops ($79\% \approx (\frac{7}{6})/(\frac{3}{2})$) while Paxos's dropped to 50 ops ($33\% = (\frac{1}{6})/(\frac{1}{2})$).

When the system is CPU-bound, increasing n requires the leader to perform more work for each client request. Since the CPU of the leader is a bottleneck for Paxos, its throughput drops as n increases. Mencius, by rotating the leader, takes advantage of the extra processing power. Figure 8(b) shows throughput for both protocols with $\rho = 0$. As n increases, Paxos's throughput decreases gradually. Mencius's throughput increases gradually because more processing power outweighs the increasing processing cost for each request. When $n = 7$, Mencius's throughput is almost double that of Paxos.

7.5 Latency

In this section, we use the three-site clique topology to measure Mencius's commit latency under low to medium load. We ran the experiments with both Nagle on and off. Not surprisingly, both Mencius and Paxos with Nagle on show a higher commit latency due to the extra delay added by Nagle's algorithm. Having Nagle's enabled also adds some variability to the commit latency. For example, with Paxos, instead of a constant commit latency of 100 ms at the leader, the latency varied from 100 to 250 ms with a concentration around 150 ms. Except for this, Nagle's algorithm does not affect the general behavior of commit latency. Therefore, for the sake of clarity, we only present the results with Nagle off for the first two experiments. With Nagle turned off, all experiments with Paxos showed a constant latency of 100 ms at the leader and 200 ms for the other servers. Since we have three servers, Paxos's average latency was 167 ms. In the last set of experiments, we increased the load and so turned Nagle on for more efficient network utilization.

In a wide-area system, the load of different sites can be different for many reasons, such as time zone. To demonstrate the ability of Mencius to adjust to a changing client load, we ran a three-minute experiment with one client on site A and one on B. Site A's client generated requests during the first two minutes and site B's client generated requests during the last two minutes. Both clients generate requests at the same rate ($\delta \in [100\text{ ms}, 200\text{ ms}]$). Figure 9 shows that during the first minute when only site A generated requests, all requests had the minimal 100 ms commit latency. In the next minute when both sites A and B generated requests, the majority of the requests still had the minimal 100 ms delay, but some requests experienced extra delayed commits of up to 100 ms. During the last minute, the laten-

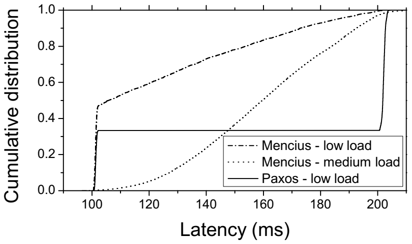


Figure 10: Commit latency distribution under low and medium load

cies return to 100 ms.

To further see the impact of delayed commit, we ran experiments with one client at each site and all three clients concurrently generating requests. Figure 10 plots the CDF of the commit latency under low load (the inter-request delay of $\delta \in [100\text{ ms}, 200\text{ ms}]$) and medium load ($\delta \in [10\text{ ms}, 20\text{ ms}]$). We show only the low load distribution for Paxos because the distribution for medium load is indistinguishable from the one we show. For Paxos, one third of the requests had a commit latency of 100 ms and two thirds had a 200 ms latency. With low load the contention level was low and delayed commit happened less often for Mencius. As a result, about 50% of the Mencius requests have the minimal 100 ms delay. For those requests that did experience delayed commits, the extra latency is roughly uniformly distributed in the range $(0\text{ ms}, 100\text{ ms})$. Under medium load, the concurrency level goes up and almost all requests experience delayed commits. The average latency is about 155 ms, which is still better than Paxos's average of 167 ms under the same condition.

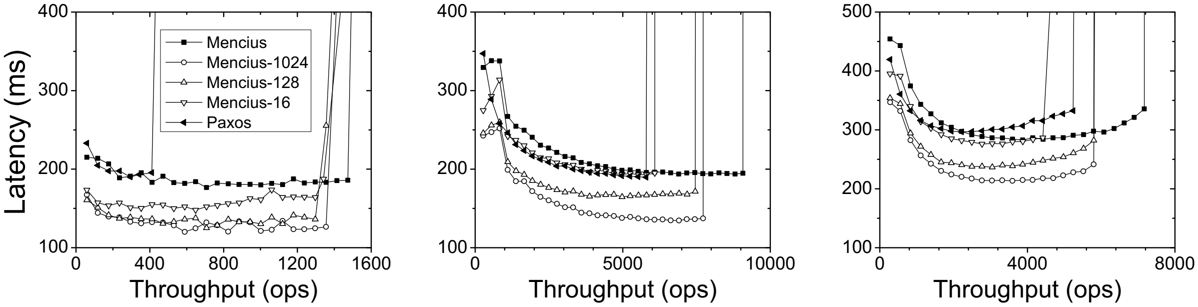
For the experiments of Figure 11, we increased the load by adding more clients, and we enabled Nagle. All curves show lower latency under higher load. This is because of the extra delay introduced by Nagle: the higher the client load, the more often messages are sent, and therefore on average, the less time any individual message is buffered by Nagle. This effect is much weaker in the $\rho = 4,000$ cases than the $\rho = 0$ case because Nagle has more impact on small messages. All experiments also show a rapid jump in latency as the protocols reach their maximum throughput: at this point, the queues of client requests start to grow rapidly.

Figure 11(a) shows the result for the network-bound case of $\rho = 4,000$. Mencius and Paxos had about the same latency before Paxos reached its maximum throughput. At this point, delayed commit has become frequent enough that Mencius has the same latency as Paxos. Lower latency can be obtained by allowing commutable requests to be reordered. Indeed, Mencius-1024, which has the lowest level of contention, had the lowest latency. For example, at 340 ops, Paxos and Mencius showed an average latency of 195 ms, Mencius-16 had an average latency of 150 ms, and Mencius-128 and Mencius-1024 had an average latency of 130 ms, which is an approximate 30% improvement. As client load increased, Mencius's latency remained roughly the same, whereas Mencius-16's latency increased gradually because the higher client load resulted in fewer opportunities to take advantage of commutable requests. Finally, Mencius-128 and Mencius-1024 showed about the same latency as client load increased, with Mencius-1024 being slightly better. This is because at the maximum client load $(1,400$ ops) and correspondent latency (130 ms), the maximum number of concurrently running requests is about 180 requests. This gave Mencius-128 and Mencius-1024 about the same opportunity to reorder requests.

Figure 11(b) shows the result for the CPU-bound case of $\rho = 0$. It shows the same trends as Figure 11(a). The impact of Nagle on latency is more obvious, and before reaching 900 ops, the latency of all four variants of Mencius increases as load goes up. This is because delayed commits happened more often as the load increased. We see the increase in latency because the penalty from delayed commits outweighed the benefits gained by being delayed, on average, for less time by Nagle. In addition, Mencius started with a slightly worse latency than Paxos, and the gap between the two decreased as throughput goes up. Out-of-order commit helps Mencius to reduces its latency: Mencius-16 (a high contention level) had about the same latency as Paxos. Finally, Mencius-128's latency was between Mencius-16 and Mencius-1024. As client load increased, the latency for Mencius-128 tended away from Mencius-1024 towards Mencius-16. This is because the higher load resulted in higher contention: increased contention gave Mencius-128 less and less flexibility to reorder requests.

In the experiment of Figure 11(c), we select delivery latencies at random. It is the same experiment as the one of Figure 11(b), except that we add a Pareto distribution to each link using the NetEm [17] utility. The average extra latency is 20 ms and the variance is 20 ms. The latency time correlation is 50%, meaning that 50% of the latency of the next packet depends on the latency of the current packet. Pareto is a heavy tailed distribution, which models the fact that wide-area links are usually timely but can present high latency occasionally. Given the 20 ms average and 20 ms variance, we observe the extra latency range from 0 to 100 ms. This is at least a twofold increase in latency at the tail. We also experimented with different parameters and distributions, but we do not report them here as we did not observe significant differences in the general trend.

The shapes of the curves in Figure 11(c) are similar to those in Figure 11(b), despite the network variance,



(a) $\rho = 4,000$, no network variance

(b) $\rho = 0$, no network variance

(c) $\rho = 0$, with network variance

Figure 11: Commit latency vs offered client load

except for the following: (1) All protocols have lower throughput despite the system being CPU-bound – high network variance results in packets being delivered out-of-order, and TCP has to reorder and retransmit packets, since out-of-order delivery of ACK packets triggers TCP fast retransmission. (2) At the beginning of the curves in figure 11(b), all four Mencius variants show lower latency under lower load because delayed commit happened less often. There is no such a trend in Figure 11(c). This happens because with Mencius we wait for both servers to reply before committing a request, whereas with Paxos we only wait for the fastest server to reply. The penalty for waiting for the extra reply is an important factor under low load and results in higher latency for Mencius. For example, at 300 ops, Mencius's latency is 455 ms compared to Paxos's 415 ms delay. However, out-of-order commit helps Mencius to achieve lower latency: Mencius-16 shows 400 ms delay while both Mencius-128 and Mencius-1024 show 350 ms delay. (3) As load increases, Paxos's latency becomes larger than Mencius's. This is due to the higher latency observed at non-leader servers. Although with Paxos the leader only waits for the fastest reply to learn a request, the non-leaders have the extra delay of FWD and LEARN messages. Consider two consecutive requests u and v assigned to instances i and $i + 1$, respectively. If the LEARN message for u arrives at a non-leader later than the LEARN message for v because of network variance, the server cannot commit v for instance $i + 1$ until it learns u for instance i. If the delay between learning v and learning u is long, then the commit delay of v is also long. Note that in our implementation, TCP causes this delay as TCP orders packets that are delivered out of order. Under higher load, the interval between u and v is shorter, and the penalty instance $i + 1$ takes is larger because of the longer relative delay of the LEARN message for instance i.

In summary, Mencius has lower latency than Paxos when network latency has little variance. The out-of-order commit mechanism helps Mencius reduce up to 30% its latency. Non-negligible network variance has negative impact on Mencius's latency under low load, but low load also gives Mencius's out-of-order commit mechanism more opportunity to reduce latency. And, under higher load, Paxos shows higher latency than Mencius because of the impact of network variance on non-leader replicas.

7.6 Other possible optimizations

There are other ways one can increase throughput or reduce latency. One idea is to batch multiple requests into a single message, which increases throughput at the expense of increased latency. This technique can be applied to both protocols, and would have the same benefit. We verified this with a simple experiment: we applied a batching strategy that combined up to five messages that arrive within 50 ms into one. With small messages ($\rho = 0$), Paxos throughput increased by 4.9 and Mencius by 4.8; with large messages the network was the bottleneck and throughput remained unchanged.

An approach to reducing latency consists of eliminating Phase 3 and instead broadcasting ACCEPT messages. This approach cuts for Paxos the learning delay of non-leaders by one communication step, and for Mencius it reduces the upper bound on delayed commit by one communication step. For both protocols, it increases the message complexity from $3n - 3$ to $n^2 - 1$, thus reducing throughput when the system is CPU-bound. However, doing so has little effect on throughput when the system is network-bound, because the extra messages are small control messages that are negligible compared to the payload of the requests.

Another optimization for Paxos is to have the servers broadcast the body of the requests and reach consensus on a unique identifier for each request. This optimization allows Paxos, like Mencius, to take full advantage of the available link bandwidth when the service is network-bound. It is not effective, however, when the service is CPU-bound, since it might reduce Paxos's throughput by increasing the wide-area message complexity.

8 Related work

Mencius is derived from Paxos [21, 22]. Fast Paxos, one of the variants of Paxos [25], has been designed to improve latency. However, it suffers from collisions (which results in significantly higher latency) when concurrent proposals occur. Another protocol, CoReFP [13], deals with collisions by running Paxos and Fast Paxos concurrently, but has lower throughput due to increased message complexity. Generalized Paxos [24], on the other hand, avoid collisions by allowing Fast Paxos to commit requests in different but equivalent orders. In Mencius, we allow all servers to immediately assign requests to the instances they coordinate to obtain low latency. We avoid contention by rotating the leader (coordinator), which is called a *moving sequencer* in the classification of Défago *et al.* [12]. We also use the rotating leader scheme to achieve high throughput by balancing network utilization. Mencius, like Generalized Paxos, can also commit requests in different but equivalent orders.

Another moving sequencer protocol is Totem [4] which enables any server to broadcast by passing a token. A process in Totem, however, has to wait for the token before broadcasting a message, whereas a Mencius server does not have to wait to propose a request. Lamport's application of multiple leaders [26] is the closest to Mencius. It is primarily used to remove the single leader bottleneck of Paxos. However, Lamport does not discuss in detail how to handle failures or how to prevent a slow leader from affecting others in a multi-leader setting. The idea of rotating the leader has also been used for a single consensus instance in the $\Diamond S$ protocol of Chandra and Toueg [10].

A number of low latency protocols have been proposed in the literature to solve atomic broadcast, a problem equivalent to the one of implementing a replicated state machine [10]. For example, Zieliński presents an optimistic generic broadcast protocol that allows commutable messages to be delivered in any order and requires $n > 3f$ [35], and elsewhere presents a protocol that relies on synchronized clocks to deliver messages in two communication steps [36]. Similar to Mencius, the latter protocol sends *empty* (equivalent to *no-op*) messages when it has no message to send. Unlike Mencius, it suffers from higher latency after one server has failed. The Bias Algorithm minimizes delivery latency when the rates at which processes send messages are known in advance [2], an assumption that Mencius does not make. Schmidt *et al.* propose the M-Consensus problem for low latency atomic broadcast and solved it with Collision-fast Paxos [32]. Instead of learning a single value for each consensus instance, M-Consensus learns a vector of values. Collision-fast Paxos works similar to Mencius as it requires a server to propose an empty value when it has no value to propose but differs in its way of handling failures: it allows a non-faulty server to take over the slot of a faulty server, which makes out-order-commit impossible when Collision-fast Paxos is used to implement atomic broadcast.

We are not the first to consider high-throughput consensus and fault-scalability. For example, FSR [16] is a protocol for high-throughput total-order broadcast for clusters that uses both a fixed sequencer and ring topology. PBFT [7] and Zyzzyva [19] propose practical protocols for high-throughput consensus when processes can fail arbitrarily. Q/U [1] proposes a scalable Byzantine fault-tolerant protocol.

Steward [3] is a hybrid Byzantine fault-tolerant protocol for multi-site systems. It runs an Byzantine fault-tolerant protocol within a site and benign consensus protocol in between sites. Steward could benefit from Mencius by replacing their inter-site protocol (the main bottleneck of the system) with Mencius.

9 Future work and open issues

The following are issues that require further work. In the interest of space, we only mention them briefly.

Byzantine failures It is not straightforward to derive a "Byzantine Mencius", because skipping, the core technique that makes Mencius efficient, is not built on a quorum abstraction. We plan to explore a Byzantine version of Mencius by applying techniques such as Attested Append-only Memory [11].

Coordinator allocation Mencius's commit latency is limited by the slowest server. A solution to this problem is to have coordinators at only the fastest $f + 1$ servers and have the slower f servers forward their requests to the other sites.

Sites with faulty servers We have assumed that while a server is crashed, it is acceptable that its clients do not make progress. In practice, we can relax this assumption and cope with faulty servers in two ways: (1) have the clients forward their requests to other sites, or (2) replicate the service within a site such that the servers can continuously provide service despite the failure of a minority of the servers.

10 Conclusion

We have derived, implemented, and evaluated Mencius, a high performance state machine replication protocol in which clients and servers are spread across a wide-area network. By using a rotating coordinator scheme, Mencius is able to sustain higher throughput than Paxos, both when the system is network-bound and when it is CPU-bound. Mencius presents better scalability with more servers compared to Paxos, which is an important attribute for wide-area applications. Finally, the state machine commit latency of Mencius is usually no worse, and often much better, than that of Paxos, although the effect of network variance on both protocols is complex.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 0546686. We would like to thank our shepherd Hakim Weatherspoon for helping us prepare the final version of the paper, the anonymous reviewers for their helpful and insightful comments, and the DETER testbed for the experimental environment.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, et al. Fault-scalable Byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, 39(5):59–74, 2005.

[2] M. Aguilera and R. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of ACM PODC*, pages 209–218, New York, NY, USA, 2000.

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

[4] Y. Amir, L. Moser, P. Melliar-Smith, et al. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, 1995.

[5] T. Benzel, R. Braden, D. Kim, et al. Design, deployment, and use of the DETER testbed. In *Proceedings of the DETER Community Workshop on Cyber-Security and Test.*, Aug 2007.

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of OSDI*, pages 335–350, Berkeley, CA, USA, 2006.

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

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of ACM PODC*, pages 398–407, 2007.

[9] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[10] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

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

[12] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[13] D. Dobre, M. Majuntke, and N. Suri. CoReFP: Contention-resistant Fast Paxos for WANs. Technical Report TR-TUD-DEEDS-11-01-2006, Department of Computer Science, Technische Universität Darmstadt, 2006.

[14] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of ACM PODS*, pages 1–7, New York, NY, USA, 1983.

[15] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[16] R. Guerraoui, R. Levy, B. Pochon, et al. High throughput total order broadcast for cluster environments. In *Proceedings of DSN*, pages 549–557, Washington, DC, USA, 2006.

[17] S. Hemminger. Network emulation with NetEm. In *Linux Conf Au*, April 2005.

[18] F. Junqueira, Y. Mao, and K. Marzullo. Classic Paxos vs. Fast Paxos: Caveat emptor. In *Proceedings of the 3rd USENIX/IEEE/IFIP Workshop on Hot Topics in System Dependability (HotDep'07)*, 2007.

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

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[21] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

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

[23] L. Lamport. Lower bounds on asynchronous consensus. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23, 2003.

[24] L. Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[25] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

[26] L. Lamport, A. Hydrie, and D. Achlioptas. Multi-leader distributed system. U.S. patent 7,260,611 B2, Aug 2007.

[27] J. Lorch, A. Adya, J. Bolosky, et al. The SMART way to migrate replicated stateful services. In *Proceedings of the ACM SIGOPS EuroSys*, pages 103–115, New York, NY, USA, 2006.

[28] J. MacCormick, N. Murphy, M. Najork, et al. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of OSDI*, pages 105–120, Berkeley, CA, USA, 2004.

[29] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. Technical Report CS2008-0930, Dept. of Computer Science and Engineering, UC San Deigo, 2008.

[30] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks, Jan. 1984.

[31] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[32] R. Schmidt, L. Camargos, and F. Pedone. On collision-fast atomic broadcast. Technical Report LABOS-REPORT-2007-001, Éole Polytechnique Fédérale de Lausanne, 2007.

[33] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, pages 299–319, Dec. 1990.

[34] J. Wensley, L. Lamport, J. Goldberg, et al. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Tutorial: hard real-time systems*, pages 560–575, 1989.

[35] P. Zieliński. Optimistic generic broadcast. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 369–383, Kraków, Poland, September 2005.

[36] P. Zieliński. Low-latency atomic broadcast in the presence of contention. *Distributed Computing*, 20(6):435–450, 2008.

[37] ZooKeeper. `http://hadoop.apache.org/zookeeper`.

Notes

[1] Mencius, or Meng Zi, was one of the principal philosophers during the Warring States Period. During the fourth century BC, Mencius worked on reform among the rulers of the area that is now China.

[2] There are other structures of state machines, such as a primary-backup structure where only one server executes the command and communicates the result to the rest of the servers, or one in which a command generates multiple responses, each sent to different clients. Our protocol can be adapted to such structures.

[3] To eliminate trivial implementations, we require that there exists an execution in which the coordinator proposes a value $v \neq$ *no-op* that is chosen as the consensus value.

[4] In practice, one non-faulty server is elected to lead the revocation process to avoid wasting resources or causing liveness problems. See [29] for more detailed discussion.