UNIVERSITY OF CALIFORNIA, SAN DIEGO

State Machine Replication for Wide Area Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Yanhua Mao

Committee in charge:

Professor Keith Marzullo, Chair
Professor Chung-Kuan Cheng
Professor Curt Schurgers
Professor Amin Vahdat
Professor Geoffrey M. Voelker

2010

The dissertation of Yanhua Mao is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2010

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my gratitude to a few people, who made all the work presented here possible.

To my father Jinhan Mao and my mother Xinmin Hu for reserving nothing in loving me, and for supporting me in any ways they can to make my life easier.

To my advisor Professor Keith Marzullo, for his patience in teaching me, for trusting my ideas, for guiding me to break my limit, for squeezing time from his department chair duty, and for being there for me especially when I need his help the most.

To Doctor Flavio Junqueira, for his helpful advices, for all the insightful discussion, and for all the hard work he put into our publications.

To my girlfriend Shengxin Tang for being by my side on the best and on the worst, for learning and growing with me as a person, and for making life much more colorful.

To my folks in China, especially Yong Wan and Yi Hu, for all the support from the homeland.

To my committee members Professor Geoff Voelker, Professor Amin Vahdat, Professor CK Cheng, Professor Curt Schurgers for dedicating a bit of their time to evaluate my research work.

To my dear friends Shan Yan, Chong Zhang, Haixia Shi, Huaxia Xia, Yuanfang Hu, Liang Jin, Jin Cai, Wanping Zhang, Yuzhe Jin, Yi Jing, Zhongyi Jin, Yi Zhu, Jing Zhu, Shaofeng Liu, Chengmo Yang, Wenyi Zhang, Min Xu, Luo Gu, Kian Win Ong, Dan Liu, Junwen Wu, Zhou Lu, Yushi Shen, Meng Wei, Ying Zhang, Zheng Wu, Dayou Zhou and Liwen Yu for making these years in San Diego very pleasant.

To the CSE professors. In particular to Professor Stefan Savage, Professor Alex Snoeren, Professor Joe Pasquale and Professor Russell Implagliazzo for all the helpful discussions.

To Marcos Aguilera for the alternate proof in section 3.3.2.

To Rich Wolski for providing us with the NWS traces section 3.3.3.

To NSF for funding my research.

Chapter 3, in part, reprints of material as it appears in “Classic Paxos vs. Fast Paxos: Caveat Emptor”, by Flavio Junqueira, Yanhua Mao, and Keith Marzullo, in the

Proceedings of the 3rd Workshop on Hot Topics in System Dependability (HotDep), June, 2007. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter 5, in part, reprints of material as it appears in "Mencius: Building Efficient Replicated State Machines for WANs", by Yanhua Mao, Flavio Junqueira, and Keith Marzullo, in the Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), December, 2008. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter 6, in part, reprints of material as it appears in "Towards Low Latency State Machine Replication for Uncivil Wide-area Networks", by Yanhua Mao, Flavio Junqueira, and Keith Marzullo, in the Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep), June, 2009 and "RAM: State Machine Replication for Uncivil Wide-area Networks", by Yanhua Mao, Flavio Junqueira, and Keith Marzullo, unpublished technical report, Department of Computer Science and Engineering, University of California, San Diego, October, 2009. The dissertation author was the primary coauthor and co-investigator of the two papers.

VITA

2002	B.Eng., Department of Computer Science and Technology, Tsinghua University, Beijing, P.R.China
2002-2004	Research Assistant, Department of Computer Science and Technology, Tsinghua University, Beijing, P.R.China
2002-2003	Teaching Assistant, Department of Computer Science and Technology, Tsinghua University, Beijing, P.R.China
2004	M.Eng., Department of Computer Science and Technology, Tsinghua University, Beijing, P.R.China
2004-2010	Research Assistant, University of California, San Diego
2010	Ph.D., Department of Computer Science and Engineering, University of California, San Diego

PUBLICATIONS

Yanhua Mao, Flavio Junqueira, and Keith Marzullo, "Towards Low Latency State Machine Replication for Uncivil Wide-area Networks", in the Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep), June, 2009.

Yanhua Mao, Flavio Junqueira, and Keith Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs", in the Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), December, 2008.

Flavio Junqueira, Yanhua Mao, and Keith Marzullo, "Classic Paxos vs. Fast Paxos: Caveat Emptor", in the Proceedings of the 3rd Workshop on Hot Topics in System Dependability (HotDep), June, 2007.

ABSTRACT OF THE DISSERTATION

State Machine Replication for Wide Area Networks

by

Yanhua Mao

Doctor of Philosophy in Computer Science

University of California, San Diego, 2010

Professor Keith Marzullo, Chair

State machine replication is the most general approach for providing highly available services with strong consistency guarantees. This dissertation studies protocols for implementing replicated state machines for wide area networks. First it demonstrates the challenges by comparing two protocols designed for local area networks in a cluster-based wide-area setting and shows that existing protocols designed for local area networks do not perform well in wide-area settings. A generic rotating leader protocol is then designed for wide-area multi-site systems. From this generic protocol, two more protocols are derived and evaluated: *Mencius* for crash failures and *RAM* for Byzantine (arbitrary) failures. Mencius has low latency because it is based on a rotating leader design that allows all servers to immediately propose client requests upon receiving them. It also has high throughput because the rotating leader design helps to balance both the computation and communication loads. RAM is designed to provide low latency while handling both uncivil rational and irrational behavior. RAM applies three techniques to reduce latency: a rotating leader design, Attested Append-only memory (A2M) and a novel *Mutually Suspicious Domains* (MSD) trust model. Uncivil rational behavior is a result of selfish local administrators who try to optimize for local utility. Such behavior

is discouraged by the protocol so that any divergence from the civil (correct) behavior only increases the probability of reducing local utility. Irrational behavior can arise from outside attack that tries to disturb the system. RAM is designed to expose as much of this kind of behavior as possible. When detection is impossible, RAM resorts to damage control methods that bound the decrease in system utility.

Chapter 1

Introduction

The most general approach for providing a highly available service is to use a replicated state machine architecture [58]. 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 [63] to asynchronous systems. Depending on the failure model adapted, this approach can also be used to tolerate a wide range of failure: from the most restricted crash failures to the most general Byzantine (arbitrary) failures.

Today, asynchronous system is popular among both researchers and practitioners due to its relaxed requirement on timing. Recent examples of services that use asynchronous replicated state machines include Chubby [13, 15], ZooKeeper [69] and Boxwood [46]. It is, however, well-known that consensus is unsolvable in a purely asynchronous environment [26], *i.e.*, no deterministic protocol can guarantee safety and liveness at the same time. The most popular approach to circumvent this impossibility result is to use unreliable failure detectors. By taking this approach, one can design protocols that are always safe despite the unbounded number of mistakes the failure detectors may make. The protocols provide liveness only during a period of synchrony, *i.e.*, when the failure detector eventually becomes accurate. A large number of protocols [14, 25, 36, 39, 47] have since been designed using failure detectors. Ω, a class of

failures detector that provide eventually accurate leader election functionality, is also proven to be the weakest failure detector for solving consensus. Leader based protocols, such as Paxos [36,37] and PBFT [14], later become the de facto standard for implementing replicated state machines: not only do they require the minimal synchrony augment for solving the problem, they are also efficient due to the use of a single leader to assign commands to consensus instances.

Despite the large number of existing protocols, most of them are designed for and deployed in local-area network. Efficiency has been one of most common concern on wide-area deployment. With the rapid growth of wide-area services such as web services, grid services, and service-oriented architectures, wide-area replicated state machine s become more and more appealing due to its strong consistency guarantees. Also, replication across multiple locations is the only way for tolerating co-location failures such as fire and power outage that can bring down all the machines in the same location at the same time. So, a basic research question is how to provide efficient and effective state machine replication in the wide area – the thesis of this dissertation.

While it is certainly possible for one to pick an application and design efficient replicated state machine protocols for such an application, doing so, however, would lose the generality of the replicated state machine approach. By not choosing a specific application, this dissertation focuses on the most fundamental properties of the problem. Doing so makes it possible for the solutions in this dissertation to be widely applicable.

Chapter 2 reviews the background materials for this dissertation, such as, asynchronous message passing model, failure model, consensus and replicated state machine, the aforementioned impossibility result, the concept of failure detectors, and the lower bounds for asynchronous consensus. We also review two de facto standard leader-based protocols: Paxos for crash failure and PBFT for Byzantine failure.

Chapter 3 is our first step toward an efficient protocol design. We start the chapter by summarizing our motivation for wide-area replicated state machines and the challenges we face. To further demonstrate the challenges, we proceed with a comparison of two protocols, namely Paxos and Fast Paxos, in a wide-area client-server environment. Our analysis and simulation results show that Fast Paxos, though designed to be faster than Paxos in a local-area environment, is actually slower in this wide-area environ-

ment. We conclude the chapter with a summary of important issues, such as contention, quorum size, and latency variances, that need to be considered when designing new protocols for wide-area systems.

In chapter 4, we move our focus to wide-area multi-site systems. While traditional leader-based protocols are efficient, they have inherent disadvantages in a multi-site system. From the point view of individual sites, the leader has unfair performance advantages over other sites. From point of view of the whole system, the leader is a single point bottleneck that limits throughput. To cope with these disadvantages, we introduce a generic rotating-leader based protocol called $\mathscr{R}$. The core mechanism of $\mathscr{R}$ is a partition of the consensus sequence space so that each site acts the default leader for a subset of the instances. By doing this, we remove the single leader bottleneck by spreading of load of being the leader among of sites. We also introduce *simple consensus* for replacing traditional consensus in the replicated state machine approach. Simple consensus is a variant of consensus that has restricted initial state: we show that it is equivalent to consensus in solvability. Together, simple consensus and the rotating-leader scheme help $\mathscr{R}$ avoid contention during a period of stability. Simple consensus also makes certain operation cheaper to implement. This feature, enhanced with some further optimizations, helps all sites to enjoy the low latency that was only possible for the leader in a traditional leader-based design. $\mathscr{R}$ was, however, not efficient in term of message complexity, to optimize which we need to find opportunities in the specific implementation of simple consensus.

Chapter 5 continues our study of wide-area multi-site system, this time, in a crash failure environment. The core result of this chapter is Mencius: an efficient crash-failure rotating-leader protocol that has high throughput under high load and low latency under low load. The protocol is derived from Paxos and uses the generic rotating-leader scheme of protocol $\mathscr{R}$. First, we derive Coordinated Paxos from Paxos to implement simple consensus efficiently. We then apply the rotating-leader scheme of protocol $\mathscr{R}$ to obtain an inefficient intermediate protocol $\mathscr{P}$. Finally, we apply a set of optimizations to protocol $\mathscr{P}$ to obtain Mencius. The derivation of Mencius makes it easier to understand its correctness and how it works. We also discuss important implementation considerations for Mencius and run experiments to evaluates its latency, throughput, scalability,

and ability to adapt to changing environment and load.

Chapter 6 extends our study to Byzantine failures. The core result of this section is a low-latency protocol RAM. RAM is built on three core techniques: (1) the **R**otating leader scheme of $\mathscr{R}$; (2) the use of **A**ttested append-only memory (A2M) for reducing latency; and (3) the assumption of **M**utually suspicious domains (MSD) that allows processes in the same site to trust each other. The development of RAM is similar to that of Mencius, and so has similar advantages. First, we introduce Coordinated Byzantine Paxos as an efficient solution to simple consensus, taking advantages of A2M and MSD. We then apply the rotating leader scheme of $\mathscr{R}$ to obtain an intermediate protocol $\mathscr{Q}$. Finally we optimize $\mathscr{Q}$ to obtain RAM. We also extend the traditional view of Byzantine failures by introducing uncivil rational and irrational behavior. These two classes of behavior have different root cause: uncivil rational behavior is caused by faulty/selfish administrators that try to game the system to improve its own utility; irrational behavior is a result of outside attack that seek to disrupt the system. Different methods are warranted to cope with these two classes of behavior. We seek to discourage uncivil rational behavior by reenforcing the protocol such that any divergence from the correct behavior can only hurt the utility of a site, so the best strategy for a uncivil rational site is to follow the correct behavior. The core mechanism for achieving this goal is revocation against uncivil sites. For irrational behavior, we also seek to expose them as much as possible through failure detection. However, not all failures can be pin-pointed. So, when detection is impossible, we resort to damage control techniques that limit the decrease of system utility. With these goals in the design, RAM is also a complex protocol. While this chapter has explained the core mechanisms that makes this approach feasible, future work is warranted for fully understand the policies to make these mechanisms effective. This chapter makes a first step toward such a goal with a prototype implementation of RAM. We use the implementation to evaluate not only the latency, throughput, and scalability of RAM, but also the implication of the revocation mechanism.

Finally, chapter 7 concludes this dissertation.

Chapter 2

Background

This chapter reviews background materials that are necessary for understanding this dissertation. Section 2.1 reviews the asynchronous message passing model and introduces both the crash failure model and Byzantine failure model. Based on the model, section 2.2 and 2.3 further introduce the consensus problem and the replicated state machine approach. Section 2.4 explains a well-known impossibility result for asynchronous consensus and section 2.5 discusses how to use unreliable failure detectors to circumvent this impossibility result. Section 2.6 summarizes important lower bounds that has been established for asynchronous consensus. Finally, section 2.7 and 2.8 review the Paxos consensus protocol for crash failures and the PBFT replicated state machine protocol for Byzantine failures.

2.1 System Model

For any distributed system, it is important to establish an appropriate system model for understanding the possible behavior of not only computation nodes but also the environment within which the nodes run. Basing on a specific model, one can then design distributed protocols to solve distributed computing problems (*e.g.*, consensus and replicated state machines for this dissertation). Since system models allow us to explore all possible behavior of a protocol, it enables us to formally prove the correctness of a protocol and analyze its efficiency and/or complexity. Moreover, the model also enables us to discuss the solvability of a distributed problem and obtain theoretical bounds

for understanding the minimal resource needed for solving a particular problem. This dissertation focuses on the *asynchronous message passing model*, a popular model for wide-area networks. To model the node behavior, this dissertation uses both the *crash failure model* and the *Byzantine failure model*. The reminder of this section gives a brief review of these topics.

2.1.1 Asynchronous message passing model

An asynchronous message passing system consists of three main component: (1) the environment such as the communication channels and adversaries, (2) the processes such as client and server computation nodes, (3) and the protocol that solves a specific distributed problem. In this model, processes are characterized as deterministic state machines (possibly with infinite states) that can communicate with each other by exchanging messages via point to point asynchronous channels.

More formally, a *message passing system* consists a collection of processes. Each *process* is characterized as a deterministic state machine that executes in a series of rounds. In each *round*, a process p receives a message m (possible empty or multiple physical messages) that is addressed to p. The protocol specifies the deterministic state machine, which takes m and its current state s as input, outputs a new state s', and generates messages to be sent to one or more processes as needed.

On top of the message passing model, system synchrony is used to model the speed of processes and message channels. *Synchronous systems* make assumptions on the upper bounds of the processing speed of processes and the deliver delay of messages. Protocols can be designed around these bounds and take timeouts as additional inputs. *Asynchronous systems* do not make such assumptions, and so can not take advantage of timeouts.

More formally, in *asynchronous systems*, no assumption is made on the processing speed of a process and how long it takes for a process to complete a round. A process may also have access to a local clock, but no assumption is made on either the accuracy or the drift of the clock. Such inaccurate clocks may not be used to guarantee safety or ensure liveness, but it is common practice to use them to provide liveness when the system enters a period of synchrony. No assumption is made on how long it takes for

the message passing channels to deliver messages, *i.e.*, there is no known upper bound on message delivery delay.

Most wide-area systems and modern operating systems are considered as asynchronous systems, even though known upper bounds do exist in such systems. The reason behind this classification is because the worst case delay can be orders of magnitude higher than the common case. For example, it typically takes a few microseconds for a process to complete a round on modern operating systems, however, several milliseconds may be needed if the system experiences a context switch or a page miss. As another example, the typical one way network delay within the North America continent is on the order of tens of milliseconds. However, when BGP routes are being updated, it may takes seconds for BGP to be stabilized and message to be delivered in the wide-area. Should we assume synchronous system and design the protocols around these extremely large and rarely occurred bounds, we would have to sacrifice the common case performance of the systems. As a result, people resort to asynchronous model and design protocols that do not rely on these bounds, as least not in the common case.

Finally, most works in the literature assume the channels only deliver messages that were sent by the processes, *i.e.*, they do not create message by themselves. However, different works may assume slightly different other properties. For example, some works assume *reliable channels* that do not drop messages whereas some only assume *fairly-lossy channels* in which the messages will be delivered eventually if the processes keep re-sending them. Some works assume an atomic broadcast primitive while others do not. Moreover, the message channel may or may not have FIFO properties. However, one can always implement reliable channels on top of fair-lossy channels and/or FIFO channels on top of non-FIFO channels [8]. Unless otherwise explicitly noted, this dissertation assumes that processes do not have atomic broadcast primitive and the channels are reliable and FIFO, mostly because we study wide-area system in this dissertation and used TCP as the transportation protocol in our implementation.

So far, we have summarized the *asynchronous message passing model*, which is a message passing system running in an asynchronous environment. Figure 2.1 summaries this model. The model is also sometimes referred as the *time-free asynchronous model* as the servers either have no access to a clock or the clock is too inaccurate to be

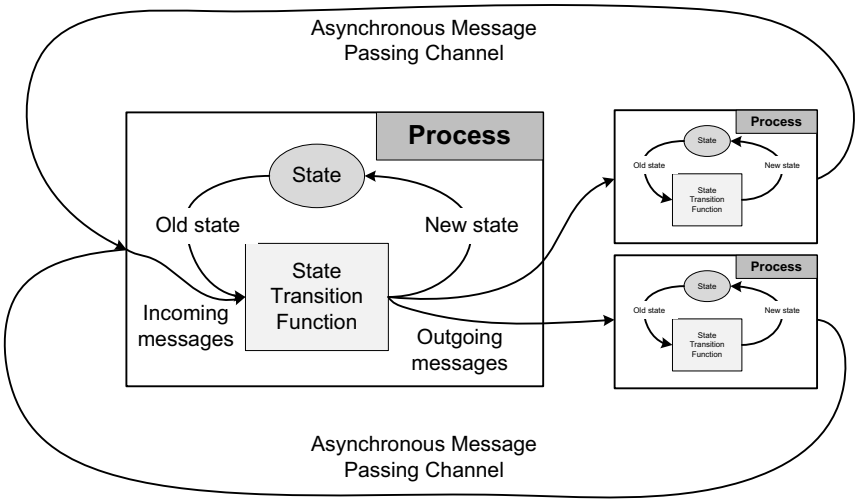


Figure 2.1: The asynchronous message passing model

of any usefulness. The asynchronous message passing model defines the possible behavior of the environment, while the model and the protocol define the possible behavior of the processes. A process fails when it diverges from this pre-defined behavior. The next section further explains two widely used failure models: the crash failure model and Byzantine failure model.

2.1.2 Failure models

A *failure* is the behavior of a process diverging from its specification defined by the system model and the protocol. Such a process is said to be *faulty*. A process is said to be *correct* if it is not faulty. Depending on the failure model, restrictions may be applied to the behavior of faulty processes. Most common failure models are the *crash* failure (fail stop) model and the *Byzantine* (arbitrary) failure model.

With the crash failure model, processes can only fail by stopping to execute any further asynchronous rounds. Usually, there is no restriction on when the failure can occur, *i.e.*, a failure can occur in the middle of a round. As a result, a process that fails when broadcasting a message to all processes may only send the message to a subset of the recipients.

By contrast, the Byzantine failure model imposes no restriction on how faulty processes may behave. An arbitrarily faulty process can stop running, omit some or all messages, or send arbitrary messages. It can even deliberately collude with other faulty processes and try to "trick" correct processes in some ways. Byzantine failure model also often assume the existence of adversaries in the environment. The adversaries can, for example, generate messages or temporarily disrupt communication to try to disturb the replicated service.

When dealing with sender failures, cryptographic primitives are often uses. For example, public/private key infrastructure can be used to authenticate and verify the messages sent by the processes. In this case, each process holds its own private key and has knowledge of the public keys of all other processes. Private keys are used to sign the messages before they are sent out and public keys are used to verify that messages are indeed sent by claimed sender. If the private key of a process is leaked or compromised, the server is consider as faulty. This is because the adversaries can now use the leaked key to assume the identity of the process and engage arbitrary communication with the rest of the processes. Processes and adversaries are also assumed to have limited computation power that they cannot subvert the cryptographic primitives. Adversaries are often assumed to be able to delay the delivery of messages, though, they cannot cause indefinite long delays.

It is also necessary for systems to restrict the number of failures that can occur for both crash and Byzantine failure models. This can be done by specify a threshold, *i.e.*, the maximum number of failures that can occur for certain type of processes, or a failure pattern, *i.e.*, all possible combinations of faulty processes that can occur. For simplicity, this dissertation only consider the threshold model, since most results for threshold model can be translated into their counterparts based on failure patterns.

2.2 Consensus

Consensus is a fundamental coordination problem that requires a group of processes to agree on a common output, based on their (possibly conflicting) inputs. A unique identity is assigned to each server, since consensus is not solvable if the servers

are indistinguishable [8].

In the context of replicated state machine, consensus input values are typically requests sent to servers from the clients. The servers then take client request and *propose* them as the initial consensus input values. They later *decide* on one of proposed value as the output.

More formally, the *consensus* problem is defined as follows: Each server in a consensus starts with a initial value to *propose* and later *decides* on some proposed value. A consensus implementation satisfies the following four properties [8].

Definition C.1 (Termination): Every correct server eventually decides on some value.

Definition C.2 (Validity): If all servers propose the same value v, then every correct server decides on v.

Definition C.3 (Integrity): Every correct server decides on at most one value.

Definition C.4 (Agreement): If a correct server decides on v, then every correct server that decides decides on v.

Of the four properties above, C.1 (Termination) is a *liveness property*, *i.e.*, it requires the processes eventually make progress. The other three are *safety property* as they must be hold at all time.

It is possible for the system to enter into a state in which the consensus outcome is determined but not yet known by any of the servers. Instead of proof such a state exists, we later show it in example protocols. We say that a value v is *chosen* when the system enters into a state in which v is only possible outcome. We say that a server *learns* value v when it knows that v is the outcome and decides on v.

Lamport [34, 37] generalize the problem by assigning roles to processes. There are *proposers* that propose values, *acceptors* that accept proposals (if certain conditions are met) and *learners* that learn the outcome of consensus. A process can take on all three roles or any combination of them. The original scenario can be considered as a special case in which all processes implement exactly all three roles. Consensus defined in this way requires the same safety and liveness properties except instead of having servers propose and decide on values, the proposers propose and the learners decided

on values. Based on this structure, Lamport proved a series of lower bounds for asynchronous consensus [34]. We will briefly review the result in section 2.6.

2.3 Replicated state machine

The replicated state machine approach [58] is one of the most general approach for providing highly available services. Assuming a deterministic service, this is done, at a high level, by replicating the state and function of the service across a set of servers, and by using an unbounded sequence of consensus instances to agree upon the sequence of commands the replicas execute.

In more detail, a replicated state machine is consist of a set of server that replicate a deterministic application service and a set of clients that issues requests to the service by sending them to one or more servers. Each server run an unbounded sequence of consensus instances. Upon receiving a request from a client, a server *submits* the request by assigning it to one of the unused consensus instances, *i.e.*, it proposes the request as the initial consensus input of that instance. While multiple servers may propose different values to the same instance, the consensus algorithm ensure that all correct servers eventually agree on a unique request for each used instance. Once a server learns the outcome of a consensus instances, it *commits* the request provided that it has learned and committed all previous consensus instances. A server commits a request by having the application service execute it. As a result, replies may be generated and sent to one or more clients.

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 application service is deterministic, then the service state will always be consistent across servers and servers will always generate consistent responses.

When running the replicated state machine, care needs to be taken to ensure that each unique client request will eventually be executed exactly once. We illustrate the problem using the following example. Consider an execution in which multiple servers may propose different requests to the same consensus instance. Only one request can be

chosen as the outcome of that instance. The rest of the requests need to be re-proposed to other unused instances. The client may also send a request to multiple servers at the same time in hope that at least some correct servers will receive the request. This may also cause duplications. When duplication occurs, the replicated state machine must not execute the same request twice.

More formally, in a *replicated state machine*, a server *submits* requests to the state machine, and later *commits* a sequence of requests that are submitted by the servers. Assuming requests are unique, a replicated state machine implementation satisfies the following liveness and safety properties [58].

Definition R.1 (Validity): If a correct server submits a request r, then all correct servers will eventually commit r.

Definition R.2 (Agreement): If a correct server commits a request r, then all correct servers will eventually commit r as well.

Definition R.3 (Integrity): Any given request r is committed by each correct server at most once, and only if r was previously submitted.

Definition R.4 (Total Order): If two correct servers p and q both commit request r_1 and r_2, then p commits r_1 before r_2 if and only if q commits r_1 before r_2.

Note that R.1 (Validity) and R.2 (Agreement) are liveness properties, whereas R.3 (Integrity) and R.4 (Total Order) are safety properties.

The most appealing factor of the replicated state machine approach is that it 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 [63] to asynchronous systems. Recent examples of services that use replicated state machines include Chubby [13, 15], ZooKeeper [69] and Boxwood [46].

2.4 FLP impossibility result

While running consensus for the replicated state machine approach is attractive for achieving fault tolerance and providing strong consistency, it is, however, well-

known as the FLP impossibility result that consensus is impossible in a purely asynchronous system where even only one process can fail. More formally, Fischer and Lynch [26] discussed a very weak form of consensus where the only values a process can propose are 0 and 1. They assumed at most one process can fail and proved that no algorithm based on the asynchronous message passing model can maintain safety at all time while also guarantees liveness (Termination) in the presence of the probability of just a single failure. The intuition behind the result is that the only way a server can tell if another server has not failed is by receiving messages from it. However, it cannot distinguish whether a server has failed or just "very slow" when receiving no messages. This makes it inherently hard for a server to decide on the consensus outcome in the presence of failure.

The impossibility result may seem discouraging, but fortunately, most practical systems are not purely asynchronous, and so the FLP result does not apply. As a result, people make additional assumptions about the system to capture the synchrony characteristic of practical system to circumvent the impossibility result. Protocols designed with these additional assumptions usually maintain safety at all time and make progress (provide liveness) when certain synchrony conditions are met. One of the most commonly used method is to augment the servers with an inaccurate local failure detector. We will briefly review this approach in the next section.

There are other ways for circumventing the impossibility result is, for example, by requiring the system to eventually synchronize. Simply speaking, with crash failure semantics, this means there is time after which no more process crashes and all messages are delivered in time. For example, Keidar and Shraer used the *eventually synchronous model* to study the overhead of consensus failure recovery [32]. We do not discuss these alternatives further in this dissertation.

2.5 Failure detectors

Due to its simplicity and modularity, the failure detector approach has gained popularity among the distributed computing community for circumventing the FLP impossibility result. Chandra and Toueg first introduced the concept of unreliable failure

detectors [17]. They showed that consensus can be solved even with unreliable failure detectors that may make an unbounded number of mistakes. They further classified failure detectors based on their *completeness* and *accuracy* properties and showed the hierarchy of failure detector classes. They also identified the weakest failure detector necessary to solve asynchronous consensus. We review failure detector related materials in this section.

2.5.1 The concept

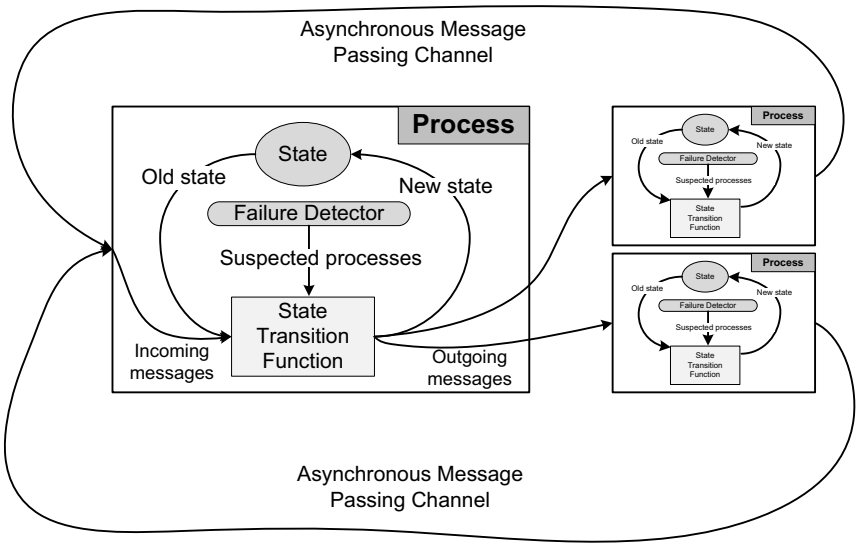


Figure 2.2: The asynchronous message passing model enhanced with failure detectors

As we have discussed, the impossibility result for asynchronous systems stem from the inherent difficulty of determining whether a server has actually failed or only "very slow". The failure detector approach augment the asynchronous model with an inaccurate external failure detection mechanism. Figure 2.2 shows the structure of the fail detector enhanced asynchronous message passing model. Each failure detector monitors the set of servers in the system and maintains a list of servers that are currently being suspected as being faulty. The failure detector is allowed to make mistakes by erroneously adding correct servers to its suspected list; for example, it may suspect that a server p has crashed even though p is still running. If the failure detector later believes

that suspecting p was a mistake, it can remove p from its list. A protocol based on failure detectors should always maintain the safety properties despite the mistakes made by the failure detectors. Usually such a protocol only guarantees progress if the failure detectors remain accurate for a sufficient long period of time.

Failure detectors are defined in term of *abstract* properties as opposed to giving specific implementations. This enables us to design protocols and prove their correctness relying solely on these properties, without referring to specific implementation or the low-level network parameters. This also makes the presentation of protocols and their correctness proof more modular.

2.5.2 Failure detector classes

Chandra and Toueg characterized failure detectors by specifying the *completeness* and *accuracy* properties. Roughly speaking, *completeness* requires that a failure detector eventually suspects all faulty processes and *accuracy* restricts the mistakes that a failure detector can make. They defined two completeness and four accuracy properties:

Strong completeness: Eventually every process that crashes is permanently suspected by *every* correct process.

Weak completeness: Eventually every process that crashes is permanently suspected by *some* correct process.

Strong accuracy: No process is suspected before it crashes.

Weak accuracy: Some correct process is never suspected.

Eventually strong accuracy: There is a time after which correct processes are not suspected by any correct process.

Eventually weak accuracy: There is a time after which some correct process is never suspected by any correct process.

Eventually strong accuracy and eventually weak accuracy are referred as *eventually accuracy* properties; strong accuracy and weak accuracy are referred as *perpetual*

Table 2.1: Eight classes of failure detectors defined in term of accuracy and completeness properties.

		Completeness	
		Strong	Weak
Accuracy	Strong	*Perfect* $\mathbb{P}$	$\mathbb{Q}$
	Weak	*Strong* $\mathbb{S}$	*Weak* $\mathbb{W}$
	Eventually strong	*Eventually perfect* $\Diamond\mathbb{P}$	$\Diamond\mathbb{Q}$
	Eventually weak	*Eventually strong* $\Diamond\mathbb{S}$	*Eventually weak* $\Diamond\mathbb{W}$

accuracy properties. Note that eventually accuracy allows failure detectors to make an unbounded number of mistakes. This reflects the inherent difficulty of determining whether a process is just slow or whether is faulty in an asynchronous system. Note that weakest accuracy property, *i.e.*, eventually weak accuracy, still requires the failure detectors to not to make certain mistakes after a certain time, which is still difficult to implement in practice. However, this is not a restriction for most practical systems: we only need failure detectors to be accurate for a sufficient long period so as that the correct servers can decide on the outcome of the consensus.

Based on the eight completeness-accuracy pairs, Chandra and Toueg further classified failure detectors into eight classes as shown in Table 2.1. For example, the class of failure detectors that have both Strong Completeness and Strong Accuracy is called the class of *Perfect failure detectors* (denoted by $\mathbb{P}$).

One can also explore the hierarchy of failure detectors using reduction technique. More formally, giving two failure detector classes $\mathscr{D}$ and $\mathscr{D}'$, $\mathscr{D}$ is said to be *reducible* to $\mathscr{D}'$ (denote as $\mathscr{D} \succeq \mathscr{D}'$) if there exist a *reduction algorithm* $T_{\mathscr{D}\to\mathscr{D}'}$ that uses $\mathscr{D}$ to maintain a variable $output_p$ at every server p such that $output_p$ is a valid output of $\mathscr{D}'$ at p for every possible execution. Clearly, $\mathscr{D}$ is stronger than $\mathscr{D}'$, as $\mathscr{D}$ can emulate the output of $\mathscr{D}'$ thus gives at least as much information about failure as $\mathscr{D}'$. Clearly, the reducible relationship is transitive. Moreover, two failure detectors $\mathscr{D}$ and $\mathscr{D}'$ are said to be *equivalent* (denoted as $\mathscr{D} \backsim \mathscr{D}'$) if and only if $\mathscr{D} \succeq \mathscr{D}'$ and $\mathscr{D}' \succeq \mathscr{D}$.

It is obvious that any failure detector has strong completeness is reducible to its weak completeness counterpart, *i.e.*, $\mathbb{P} \succeq \mathbb{Q}$, $\mathbb{S} \succeq \mathbb{W}$, $\Diamond\mathbb{P} \succeq \Diamond\mathbb{Q}$, and $\Diamond\mathbb{S} \succeq \Diamond\mathbb{W}$. Chandra and Toueg also gave an algorithm to transform any weak completeness failure

detector to its strong completeness counterpart, making weak completeness and strong completeness equivalent. Therefore, we have $\mathbb{P} \backsim \mathbb{Q}$, $\mathbb{S} \backsim \mathbb{W}$, $\Diamond\mathbb{P} \backsim \Diamond\mathbb{Q}$, and $\Diamond\mathbb{S} \backsim \Diamond\mathbb{W}$. This reduces the number of failure detector classes needs to be studied from eight to four.

Chandra and Toueg further examined the hierarchy of failure detector classes. An example algorithm was given to solve crash-failure consensus using $\mathbb{W}$ where up to $n-1$ servers can crash. Another algorithm was shown to crash-failure solve consensus using $\Diamond\mathbb{W}$ when the majority of the processes are correct ($f < \lceil n/2 \rceil$). A lower bound was proved to show that consensus cannot be solved using $\Diamond\mathbb{P}$ in crash-failure asynchronous systems with $f \geq \lceil n/2 \rceil$. Since $\Diamond\mathbb{P}$ is strong than $\Diamond\mathbb{P}$, $\Diamond\mathbb{P}$ can be used to solve asynchronous consensus with $f < \lceil n/2 \rceil$, so this lower bound is tight. This also shows that failure detectors that have perpetual accuracy properties can solve more problems that failure detectors with only eventually accuracy, therefore failure detectors with perpetual accuracy are strictly stronger.

2.5.3 The weakest failure detector for solving Consensus

The existence of the failure detector hierarchy raises the natural question: what is the weakest failure detector for solving consensus? To answer this question Chandra *et al.* [16] introduced a new failure detector class Ω. Instead of outputting a list of suspected servers, each server's failure detector outputs (trusts) a single server that is believed to be correct. Ω is a class of failure detector that there is a time after which all correct servers always trust the same server and the identity of the trusted server does not change thereafter. Ω is also called the *leader election failure detector*. On one hand, protocols, *e.g.*, Paxos, can use Ω to implement crash-failure asynchronous consensus when the majority of the servers are correct. On the other hand, it is shown that that for any failure detector that can solve consensus, the consensus protocol can be used to emulate Ω. Combine these two result, we conclude that Ω is the weakest failure detector for solving consensus. It is also possible to reduce from Ω to $\Diamond\mathbb{W}$. So, $\Diamond\mathbb{W}$ is also the weakest failure detector for solving consensus. The generic reduction from Ω to $\Diamond\mathbb{W}$ in [16] is complicated. Chu [18] also gave two simpler reductions from Ω to $\Diamond\mathbb{W}$. Both Ω and $\Diamond\mathbb{W}$ are widely used in the literature when designing consensus algorithms as

they are the minimum augments necessary for solving asynchronous consensus. Ω is particularly popular due to its simple leader-election semantic.

2.6 Lower bounds

So far, we have focused our discussion on the solvability of consensus. In this section, we further review how fast asynchronous consensus can be achieved.

Lamport studied the speed of crash failure consensus by analyzing the minimal number of communication steps needed for reaching consensus in the context of *proposers*, *acceptors* and *learners* [34]. The steps are counted from the time a value is proposed by the proposers to the time a value is learned by the learners. Since asynchrony can prevent any algorithm from reaching consensus for arbitrarily long period of time, any lower bounds obtained refer only to synchronous runs (in the contexts of failure detector enabled protocols, this means that failure detectors are accurate during these runs). Lamport showed that the answer is *three*, *two*, or *one*, depending on how many acceptors are used, how many acceptors may fail, exactly who take on the role of proposers and learners, and whether or not proposals are issued concurrently. The result can be informally summarized as follows. For an algorithm that uses n acceptors, and that can always make process despite up to f processes failures, and that works with arbitrary proposers and learners assignment:

- Learning is possible in three messages delays iff $n > 2f$
- Learning is possible in two message delays when up to e acceptors can fail, for $0 \leq e \leq f$ and $f > 0$, iff $n > 2e + f$. However, concurrent proposals can prevent an algorithm from achieving such fast learning.
- Learning is impossible in one message delay.

Note that we have discussed that $n > 2f$ is the minimal replication required for solving asynchronous consensus with Ω. The above result can be roughly translated into: when using Ω and the minimal number of replicas, three communication steps are needed for reaching consensus.

Lamport also gave special cases in which none of these bounds holds. However, those special cases are uninteresting in practice. For example, learning is possible in one round if there is only one acceptor and the acceptor process also implements a learner. With this setup, the failure of the single critical acceptor is equivalent to the total failure of the system.

2.7 Paxos

Paxos [36, 37] is an efficient asynchronous consensus protocol that tolerates up to f crash failures with $2f + 1$ replicas. Table 2.2 summaries the messages in Paxos. Processes in Paxos have roles: there are proposers that propose values, acceptors that accept proposals and learners that learn the consensus outcome based on values accepted by the acceptors. It is common for a Paxos server to take on all of the three roles.

Table 2.2: Summary of messages in Paxos

Message	Meaning
NEW-LEADER	New leader starts Phase 1
ACK	Replicas acknowledges NEW-LEADER message
REQUEST	A request is sent to the leader
PROPOSE	The leader propose a request
ACCEPT	The replicas accepts the proposed request
LEARN	Leader inform the replicas the chosen value

Paxos is a leader-based protocol: a distinguished server (proposer) is acts differently than the others and is responsible for assigning proposals to consensus instances. There can be more than one leader at the same time, but during such periods the protocol may not make progress.

Figure 2.3 illustrates the message flow in a run of a sequence of Paxos instances during failure free-executions. 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

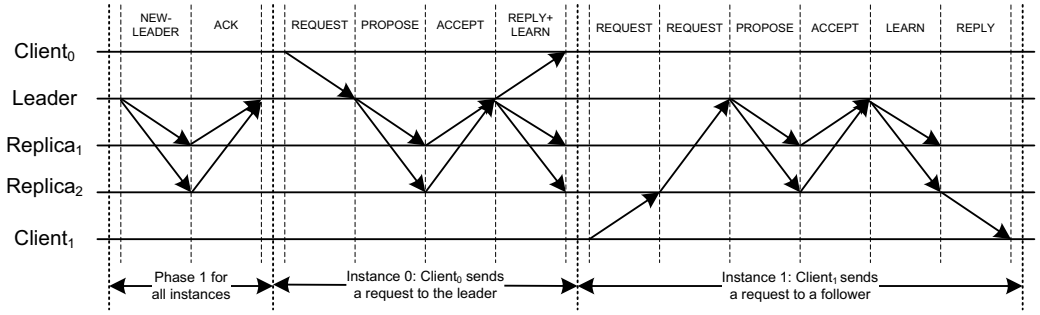


Figure 2.3: The message flow of Paxos during failure-free runs

choose a command. During normal execution all client requests go through the leader either by being sent directly to the leader or by being forwarded to the leader via another server. Upon receiving the request, it starts Phase 2 by sending PROPOSE messages 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. Figure 2.5 shows the message flow of this alternative. This option, however, increases the number of messages significantly and so can lower throughput.

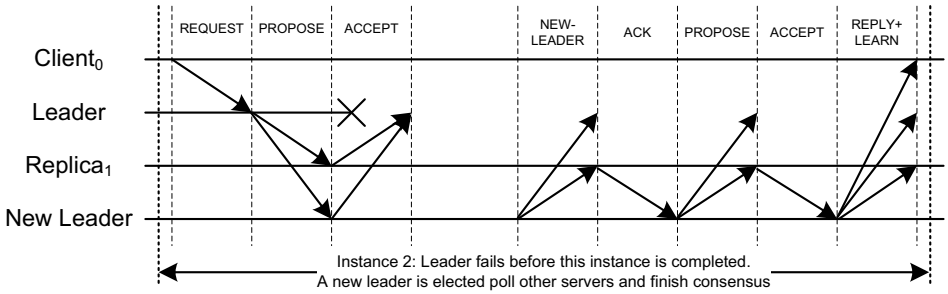


Figure 2.4: The message flow of Paxos leader change

When a leader crashes (Figure 2.4), the crash is eventually suspected, and an-

other 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 NEW-LEADER 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.

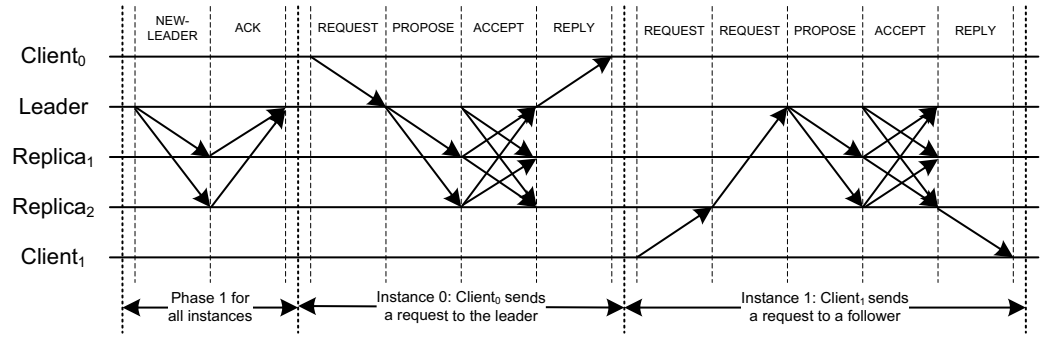


Figure 2.5: The message flow of Paxos when ACCEPT messages are broadcasted

In the steady state, it takes three communication steps ($client \rightarrow leader \rightarrow acceptor \rightarrow learner$) for a learner in Classic Paxos to learn the value if the request is directly sent to the leader.

2.8 PBFT

Table 2.3: Summary of messages in PBFT

Message	Meaning
REQUEST	A client sends the request to the leader
PRE-PREPARE	The leader proposes the client request
PREPARE	The eplicas confirm the request proposed by the leader
COMMIT	The replicas accept the confirmed proposal
REPLY	The replicas sends the reply to the client
NEW-VIEW	The replicas enter a new view

PBFT [14] is a replicated state machine protocol that tolerates up to f Byzantine failures with $3f+1$ replicas. Table 2.3 summarizes the messages in PBFT. PBFT can be considered as an extension of Paxos from crash failure model to Byzantine failure model. Indeed, they share many important characteristics: they are both leader (primary in PBFT terminology) based protocols; they are always safe and are live as long as the leader is correct; when the leader failures, they relies on leader change (view change in PBFT terminology) to provide liveness.

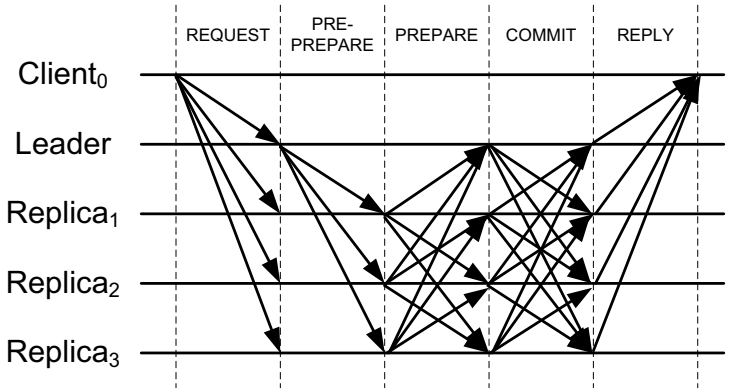


Figure 2.6: The message flow of PBFT in the steady state.

Figure 2.6 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. Note that PRE-PREPARE and PREPARE in PBFT serve similar function as PROPOSE in Paxos: the goals are to propose the request to the replicas. Similarly, COMMIT in PBFT and ACCEPT in Paxos are both

used for replicas to inform each other the values they just accepted. When a quorum of replicas ($f+1$ for Paxos and $2f+1$ for PBFT) have accepted the proposal, consensus is reach and subsequently learned.

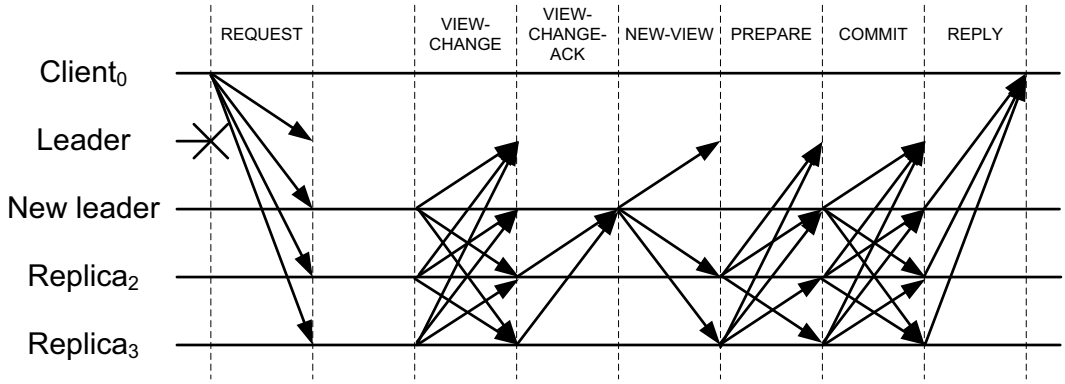


Figure 2.7: The message flow of PBFT view change

Figure 2.7 shows the message flow of PBFT when view change occurs. When the leader in PBFT fails, it is eventually suspected by other replicas since the protocol is not making progress. Correct servers then abandon the current view v and its leader l_v by changing to view $v+1$. The servers does this by broadcasting VIEW-CHANGE messages and then use VIEW-CHANGE-ACK messages to relay VIEW-CHANGE messages to server l_{v+1}, *i.e.*, the leader of view $v+1$. VIEW-CHANGE and VIEW-CHANGE-ACK messages serve similar function to the ACK messages in Paxos: to inform the new leader the current states of the replicas. Based on the information in the VIEW-CHANGE and VIEW-CHANGE-ACK messages, the new leader then computes the possible consensus outcome to propose. It does this by sending the NEW-VIEW message, which serves similar function to the PRE-PREPARE message. PBFT then proceeds with PREPARE and COMMIT messages to finish consensus, much like the Paxos resumes Phase 2 after a leader change.

Chapter 3

Benefits and challenges of wide-area networks

This chapter explains the motivations of building efficient replicated state machine protocols for wide-area networks. Section 3.1 discusses the benefits of running replicated state machine in wide-area networks and section 3.2 explains the challenges of doing so. Section 3.3 further demonstrates the challenges by comparing two crash failure replicated state machine protocols in one particular wide-area setting. Section 3.4 concludes this chapter by discussion important factors that need to be taken into considerations when designing a protocol.

3.1 Benefits

The state machine replication approach is an important tool for building highly available services and providing strong consistency. It has been deployed in local area networks as an infrastructure for coordinating applications [13,46,69]. While few wide-area systems have deployed replicated state machine the same principle can be applied for coordinating wide-area applications. It is even more beneficial to do so in wide-area networks as larger variance of message latencies in such systems makes it more difficult to coordinate applications.

Another benefit of running replicated state machine in wide-area networks is to improve service reliability and availability by increasing independent failures. To

Table 3.1: Four categories of wide-area replicated state machine systems

		Distribution of servers	
		LAN	WAN
Distribution of clients	LAN	{L×L}	{L×W}
	WAN	{W×L}	{W×W}

provide highly available services using less reliable replicas, replicated state machine relies on the assumption that replicas fail independently. Running all the replicas in the same local-area network, however, makes the system vulnerable to co-located failures such as fire, power outage, and natural disaster. For example, an electrical explosion and subsequent fire at a data center caused hours of down time to the whole data center [48]. As a result, thousands of servers in the data center were offline during that period until power was restored. Any services replicated within a local-area network such as a data center will be render unavailable in face of the aforementioned failures – all of the replicas are subject to the same failure at the same time. The only way to deal co-location failures is to distribute the replicas across multiple locations such that they will more likely to fail independently in the face of such failures.

3.2 Challenges

Despite the benefits, few systems have deployed replicated state machine in the wide-area systems. One of the most important concerns is performance. For example, the PNUTS [21] system has opted to adapt weaker consistency model due to latency concerns. The increased communication delay and decreased bandwidth of wide-area network make it more too important to deploy highly efficient protocols.

The complexity of wide-area network is also not making the problem easier. A wide-area system is more complex than a system that is simply not local-area. To illustrate that, we roughly classify distributed replicated state machine systems using two criteria: (1) how are clients distributed? and (2) how are servers distributed? Combining these two criteria, we can roughly classify systems into four categories as showed in table 3.1.

- **{L×L}:** A system whose clients and servers are all in the same local area network. This is local-area system. Such is a system is not the main focus of this dissertation.

- **{W×L}:** A system whose clients are distributed in the wide-area network, but the servers are in the same local-area network. For example, in such as system, a service is replicated in a data center and remote clients access the service via wide-area communication. In section 3.3, we compare two consensus protocols in such a {W×L} setting to illustrate some of the challenges in designing replicated state machine protocols for wide-area systems.

- **{W×W}:** A system whose clients and the servers are both distributed in the wide-area network. A multi-site system is {W×W}. Such a system is consist of several sites and each site hosts one server and several clients. In chapter 4,5 and 6, we design replicated state machine protocols for such multi-site systems.

- **{L×W}:** A system whose the clients area in the same local area network, but the servers are distributed in the wide-area network. Though interesting, such a system is outside the scope of this dissertation.

It is not difficult to imagine that protocols designed primarily for one particular network setting may not work as efficient when deployed in another setting. To further illustrate this idea, in the next section, we compare two protocols: Paxos and Fast Paxos (a varient of Paxos that is designed to have lower latency than Paxos in local area networks) in {W×L} settings and show that Fast Paxos is actually slower.

3.3 Paxos vs. Fast Paxos

Paxos and Fast Paxos are two protocols that are the core of efficient implementations of replicated state machines. In runs with no failures and no conflicts, Fast Paxos requires fewer communication steps for learners to learn of a request compared to Paxos. However, there are realistic scenarios in which Paxos has a significant probability of having a lower latency. This paper discusses one such scenario with an analytical comparison of the protocols and simulation results in a {W×L} setting.

Table 3.2: Summary of Paxos and Fast Paxos protocols.

	Paxos	**Fast Paxos**
Communication steps	3	2
Number of replicas	$2f+1$	$3f+1$
Quorum size	$f+1$	$2f+1$

3.3.1 Fast Paxos

We have already explained Paxos, an asynchronous consensus protocol that tolerates up to f failures with $2f+1$ replicas. It is also a leader based protocol: the clients send their requests to the leaders to be proposed. In the steady state, it takes three communication steps ($client \rightarrow leader \rightarrow acceptor \rightarrow learner$) for a learner in Paxos to learn the value.

Fast Paxos [39] is a variant of Paoxs that is designed to have lower latency in local area networks, hence the name. It saves one communication step by allowing clients to directly propose values to the acceptors ($client \rightarrow acceptor \rightarrow learner$). However, to preserve safety, a larger quorum of acceptors is necessary. The quorum size of Fast Paxos is $2f+1$, *i.e.*, a learner needs to know that $2f+1$ acceptors have accepted the same value for it to know consensus has been reached. This is f replicas larger than the $f+1$ quorum size of Paxos. As a result, Fast Paxos requires at least $3f+1$ acceptors while Paxos requires only $2f+1$ acceptors. In addition, Fast Paxos can suffer from *collisions*, which can happen when two or more clients send proposals at nearly the same time, and acceptors receive these proposals in different orders. Collision causes additional latency for Fast Paxos and the extra latency can be large in some cases. For simplicity, we do not consider collisions when comparison Paxos and Fast Paxos. Doing so favors Fast Paxos. Our result shows that Fast Paxos is slower despite such favoritism.

Table 3.2 summarizes some of the facts about Paxos and Fast Paxos.

3.3.2 Protocol Analysis

Fast Paxos enables a learner to learn a new client request in two communication steps, whereas Paxos requires three. In this section, we analyze the message latency of

both flavors of Paxos, and we only discuss the case in which at most one server is faulty at any time, *i.e.*, $f = 1$. In this case, Paxos requires a minimal of three servers while Fast Paxos requires four. We first introduce some notation. Let $lt(p_1, p_2)$ be the latency of a message sent from p_1 to p_2 and $pc(p_1)$ be the processing time of an operation in process p_1. We also use $smin\{A,B,C\}$ to denote the second smallest value among A,B, and C, and $tmin\{A,B,C,D\}$ to denote the third smallest value among A,B,C, and D.

We use the expression *learning latency* to denote the time for a given learner to learn a new request. For Paxos, the learning latency is given by the following expression:

$$learn_p = lt(Client, Leader) + pc(Leader) + smin\{A_1, A_2, A_3\}$$

$$A_i = lt(Leader, Acceptor_i) + pc(Acceptor_i) + lt(Acceptor_i, Leanrer), i \in \{1,2,3\}$$

For Fast Paxos, the equivalent expression is as follows:

$$learn_{fp} = tmin\{A_1, A_2, A_3, A_4\}$$

$$A_i = lt(Client, Acceptor_i) + pc(Acceptor_i) + lt(Acceptor_i, learner), i \in \{1,2,3,4\}$$

If we assume that the processing times are negligible, then we can simplify the previous equations to the following:

$$learn_p = lt(Client, Leader) + smin\{A_1, A_2, A_3\}$$

$$A_i = lt(Leader, Acceptor_i) + lt(Acceptor_i, Leanrer), i \in \{1,2,3\}$$

$$learn_{fp} = tmin\{A_1, A_2, A_3, A_4\}$$

$$A_i = lt(Client, Acceptor_i) + lt(Acceptor_i, learner), i \in \{1,2,3,4\}$$

To give an example in which Fast Paxos has a significant probability of having a higher latency compared to Paxos, we consider a $\{W \times L\}$ setting, in which all servers are in the same site, interconnected through a local-area network, and the clients are in different networks, connected to the servers through a wide-area network. Figure 3.1 shows the structure of such a system. We can then assume that the network communication among the servers is negligible compared to the cost of the wide-area latencies.

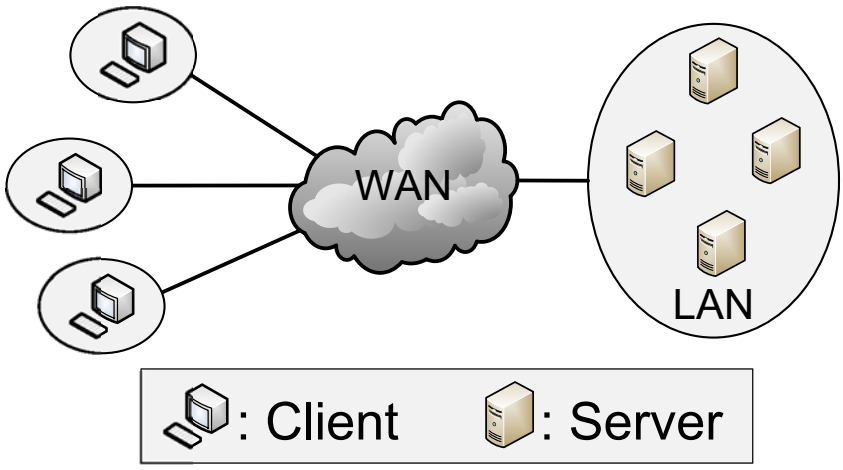


Figure 3.1: The structure of a {W×L} system

For Paxos, we then have that the time for a learner to learn that a client request has been accepted is given by:

$$learn_p = lt(Client, Leader)$$

For Fast Paxos, it is given by:

$$learn_{fp} = tmin\{lt(Client, Acceptor_i) : i \in \{1,2,3,4\}\}$$

Theorem 3.1: Suppose that wide-area latency ranges from α to β and follows a continuous probability density function (PDF) $D(x)$, where $x \in [\alpha, \beta]$. Paxos has a fixed probability of 0.6 for being faster than Fast Paxos.

Proof. The cumulative distribution function (CDF) $C(x)$ of $D(x)$ captures the probability for a message to be delivered within time x, *i.e.*, $C(x) = \Pr(latency < x)$, where $x \in [\alpha, \beta]$.

The distribution of the learning latency for Paxos is the same as the wide-area message latency distribution, as it is dominated by the message from the client to the leader and the communication among the servers is negligible. The PDF and CDF for Paxos are then given by $D_p(x) = D(x)$ and $C_p(x) = C(x)$ respectively.

Assuming that the message latencies for Fast Paxos messages are independent, the CDF of the learning latency for Fast Paxos is as follows:

$$C_{fp}(x) = \Pr(learn_{fp} < x) = 4C^3(x)(1 - C(x)) + C^4(x)$$

Recall that the probability of Fast Paxos learning a value by time x is the probability of at least three out of four messages from the client to the servers being delivered within time x. The PDF is given by:

$$P_{fp}(x) = \frac{\mathrm{d}C_{fp}(x)}{\mathrm{d}x}$$

Now suppose an independent run of both protocols. The probability P that Fast Paxos is slower than Paxos is:

$$P = \Pr(learn_{fp} > learn_p) = \int_{x=\alpha}^{\beta} P_{fp}(x)C_p(x)\mathrm{d}x$$

Noting that $C(\alpha) = 0$ and $C(\beta) = 1$, it is relatively simple to expand this equation and obtain:

$$P = 12\int_{C(x)=0}^{1} (C^3(x) - C^4(x))\mathrm{d}C(x) = \frac{3}{5}$$

□

This result implies that, independent of the distribution for wide-area message latencies, Paxos is faster than Fast Paxos for 60% of the time in this particular network topology. Note that this proof only holds for continuous distributions. For example, should the message latency is constant, the result clearly does not hold.

We offer the following explanation for the result. Intuitively, Fast Paxos has to wait for three messages out of four to make progress whereas Paxos only requires one particular message. Even though the one message for Paxos can be slow, this message is faster in most cases compared to waiting for three out of four messages.

An alternate way of obtaining this same result, but under different constraints is the following.

Theorem 3.2: Assume that the probability distribution of network latencies is not concentrated on any value. Paxos has a fixed probability of 0.6 for being faster than Fast Paxos.

Proof. This assumption is weaker than assuming that the PDF is continuous, and it implies that if we sample the distribution multiple times, then all samples will be distinct with probability 1. Fast Paxos samples the network latency distribution four times. Let these samples have values A, B, C, D, where $A < B < C < D$. Paxos samples the network

latency distribution once. Let that sample have value E. By assumption, these five samples are distinct. There are five possible relations between the value E with respect to the other four values:

1. $E < A$;
2. $A < E < B$;
3. $B < E < C$;
4. $C < E < D$;
5. $D < E$.

Each case has the same probability because we draw A, B, C, D, E from the same distribution. Thus, the probability of Cases 1, 2 or 3 is $3/5$. These are the cases when Paxos is faster than Fast Paxos. □

We can extend Theorem 3.2 to a system that tolerates up to f failures. In such a system, Paxos requires at least $2f+1$ replicas and Fast Paxos requires at least $3f+1$ replicas.

Theorem 3.3: Assume that the probability distribution of network latencies is not concentrated on any value. Paxos with $2f+1$ replicas has a fixed probability of $\frac{2f+1}{3f+2}$ for being faster than Fast Paxos with $3f+1$.

Proof. The proof is similar to that of Theorem 3.2. We draw one variable P for Paxos and $3f+1$ variables (F_0 to F_{3f}) for Fast Paxos. We order P and $F_0 - F_{3f}$. There are a total of $3f+2$ cases, in $2f+1$ of which Paxos is faster. Since each of the $3f+2$ cases also has the same probability, the probability of Paxos bing faster is $\frac{2f+1}{3f+2}$. □

From Theorem 3.3, as f increases, the probability of Paxos being faster approaches $\frac{2}{3}$.

3.3.3 Simulation

The performance of both variants of the Paxos algorithm depends upon how fast the network delivers messages to receivers. Their relative performance depends strongly on the variance of message latency. In many real networks, the variance in message latency is high due to traffic variations and non-deterministic scheduling of processes in a single computer. Informally, this observation implies that most of the time messages are delivered fast, but occasionally messages take one or two orders of magnitude more to be delivered.

In this section, we present simulation results on the latency of Paxos and Fast Paxos. The simulator we used assumes that processing time is negligible compared to message latencies, and consequently the learning latency is the sum of the message latencies. To simulate message latencies, we considered traces obtained with NWS (Network Weather Service [65]) in the GrADS testbed [11] over the period between August and October of 2002. These are traces of TCP connections between pairs of machines. Each trace contains the time to establish a TCP connection, send four bytes, receive four bytes, and close the connection.

Our simulator is trace-driven. We associate the history between two computers in our dataset with a channel of our simulator, and for every message that crosses the channel, we obtain the latency for this message from the associated history. Also, we consider failure-free runs only, and we assume no conflicts between different clients. Failure-free runs should be the common case in many systems, and collisions introduce extra complexity into our environment not necessary to make our point. In fact, had we considered collisions for Fast Paxos, the latency for Fast Paxos would have been higher.

The case we present consists of a client and a set of servers implementing Paxos, where the client is in one site and all the servers are in another site. That is, only the communication between the client and the servers crosses a wide-area network. For this scenario, we have selected two different sites A and B from our dataset, and used the traces between two machines in different sites, one in A and one in B, and between pairs of machines in site B.

Figures 3.2 and 3.3 shows the cumulative fraction of requests (y-axis) with a given latency (x-axis). The latency of Figure 3.2 includes the time for a client to send a

request to the servers implementing Paxos, the time for servers to exchange messages, and the time for a learner to learn this request by receiving accept messages from a quorum of acceptors. In addition to the latency mentioned for the case of Figure 3.2, the latency of Figure 3.3 also includes the time to send a response back to the client.

In these figures, if we draw a vertical line at some value of x_0, then the two y values of the two points in which this line crosses the curves correspond to the fraction of instances that Paxos and Fast Paxos obtain a latency value $x \leq x_0$. From the learning curves, for values of $x_0 < 90ms$, the fraction of instances for which Paxos obtain this latency is larger compared to the same fraction for Fast Paxos. The curves cross roughly at $90ms$ and, for values of $x_0 > 90ms$, the roles change, and the fraction of instances that have latency x or smaller is higher for Fast Paxos. The intuition for this result is as follows. Suppose we pick an instance of Paxos as a reference, and consider the latency for a client request to reach the proposer. If the latency for this message is low, then there is a high probability that an instance of Fast Paxos using the same latency distribution is higher. This is due to the variance in message latency. As there are more messages from the client to the acceptors, the probability that at least two messages have a higher latency is significant compared to the request message to the proposer in Paxos. If the request latency in Paxos is high, then there is a high probability that Fast Paxos is faster because it can discard one message among all four sent to the acceptors if this message is too slow.

Note also that the difference between Fast Paxos and Paxos is more noticeable in the learning latency graph. For example, if we pick the value $x = 80ms$, the difference between the fraction of instances that have at most this learning latency value is over 0.7. In the client latency figure, the difference between y fraction values for the same latency value x is not greater than 0.5. This is due to addition of the latency to respond to the client in the client latency graph, which takes another wide-area communication step. This new step increases the variability in the latency of instances as now instances that are learned fast have a non-negligible probability of having slow wide-area messages on the way back to the client.

From a different perspective, we can also draw horizontal lines to determine the latency values for which we obtain a particular fraction of instances. For example, if

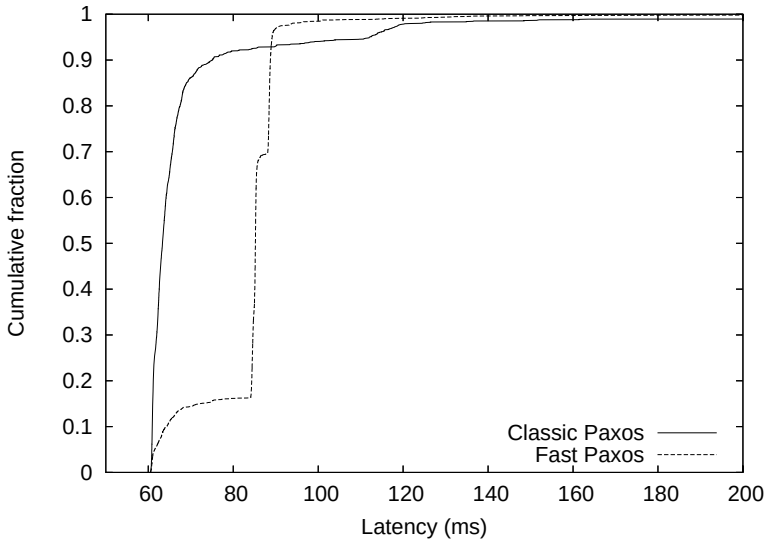


Figure 3.2: Cumulative distribution comparing Paxos and Fast Paxos, learning latency.

$y = 0.5$, then in the learning latency graph Paxos obtains this fraction with latency $62ms$, whereas Fast Paxos obtains this fraction with $83ms$. In general, for values of $y < 0.9$, the latency value for such a fraction is smaller for Paxos in both graphs. Paxos and Fast Paxos swap roles for $y > 0.9$.

3.3.4 Summary

So far, we have compared Paxos and Fast Paxos in $\{W \times L\}$ settings. Though we did not consider collisions, which favors Fast Paxos, our analytical and simulation result showed that Fast Paxos can be significantly slower in term of latency than Paxos. We also examined that the higher latency with Fast Paxos in this $\{W \times L\}$ setting is due to its larger quorum size and the variance of message deliver latency in the wide-area.

Our comparison serves as a clear example that protocol (*e.g.*, Fast Paxos) designed to perform well for one particular network setting (*e.g.*, $\{L \times L\}$) may perform poorly in another setting (*e.g.*, $\{W \times L\}$). In the next section, we conclude this chapter by discussing some of the important design considerations for efficient replicated state

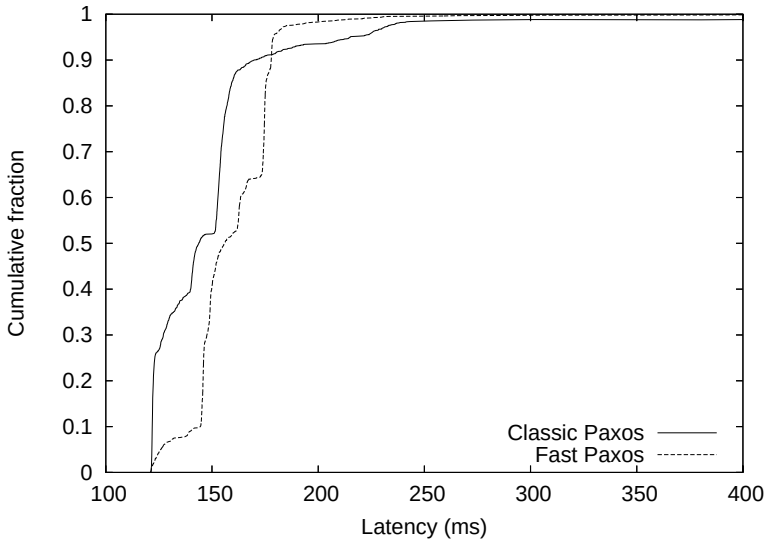


Figure 3.3: Cumulative distribution comparing Paxos and Fast Paxos, client latency.

machine protocols.

3.4 Design considerations

Theoretically, counting the number of communication steps to learn the value is a simple way to evaluate the performance of consensus protocols. Indeed, in a system where system load is low and network delivery variance is small, communication steps can be an excellent predictor of latency. However, communication steps do not always translate into practical performance because of the asynchronous nature of the system. Especially in a wide-area network, there can be a large variance in message latency. In addition, often theoretical analysis overlooks other factors that can add to latency, such as available network bandwidth and operating system scheduling. In this section, we discuss a set of issues to consider when designing a consensus protocol for a given system.

Message complexity: Message complexity plays an important role in high-load systems. An increase in the number of messages by a constant factor can have signif-

icant performance penalty when system load increases. For example, Dobre *et al.* propose a new consensus algorithm that performs better under collisions [25], but requires a larger number of messages by a constant factor. Not surprisingly, their evaluation results show that this larger number of messages can cause a significant performance penalty when system load is high.

Quorum size: Larger quorum sizes require higher degrees of replication for the same degree of fault tolerance. In addition, in systems with significant variance in message delay and in operating system scheduling latency, a larger quorum size can lead to significantly higher latency [31].

Single critical path: The existence of single critical path makes performance vulnerable to a single performance problem along that path. Increased delay from the client to the leader in Paxos slows the whole algorithm down. Paxos can achieve up to 50% performance gain when it is able to dynamically change the leader [56].

Resistance to collision. Fast learning consensus protocols, such as Fast Paxos, rely on the absence of collisions to achieve high performance, and according to the Collision-Fast Learning Theorem by Lamport no general consensus algorithm can be fast learning upon collisions [34]. Collision is inherently hard to avoid in wide-area environments, in particular when there are multiple clients submitting requests concurrently. Because of large values and high variance with message latency in wide-area environments, there is often a higher probability of collisions when using larger quorums such as the quorums that Fast Paxos requires.

Besides designing new protocols. The following two techniques can be used to achieving high performance across a wide range of environments, though not without their respective drawbacks.

1. Run multiple protocols concurrently;
2. Switch between protocols depending on the network conditions.

For example, Fast Paxos outperforms Paxos when both system load and message delay variance are low, but can do worse otherwise [25]. One can hope to achieve the

better of two protocols at any time by using both protocols. However, this may not be ideal in practice. When two protocols are run at the same time, additional care need to be taken to make sure the two protocols choose consistent value. In addition, more messages will be sent when running the two at the same time, which may hurt the system performance as load increases.

Putting aside the policy of deciding when to switch, switching between protocols is straightforward when the consensus protocol is used to support replicated state machines. Switching can be learned as a proposal in the command sequence of the replicated state machine. A system is usually designed to have a maximum number (say α) of concurrent consensus instances, and so if a switching command is learned in instance i, switching can be achieved in instance $i+\alpha$. Furthermore, a *no-op* (a command that leave the state unchanged and generates no replies) can be used in bulk to fill the gap from instance $i+1$ to $i+\alpha-1$ so as to speed up the switching process. The less straightforward question is when to make the switch, which usually ties with the network settings and the goal for which the system tries to optimize.

3.5 Summary

Running replicated state machine s in wide area networks can benefit applications in the wide-area. It also helps to cope with co-locations failures. Doing so is also challenging: most existing protocols are designed for local area networks and may not work well in wide-area systems. Designing an efficient protocol for wide-area system needs take into account not only the complex system structure but also the high message deliver variance in the wide area network. In the next three chapters of this dissertation, we continue studying replicated state machine protocols in one such setting: multi-site $\{W\times W\}$ systems.

3.6 Acknowledgement

This chapter is, in part, reprints of material as it appears in “Classic Paxos vs. Fast Paxos: Caveat Emptor”, by Flavio Junqueira, Yanhua Mao, and Keith Marzullo, in

the Proceedings of the 3rd Workshop on Hot Topics in System Dependability (HotDep), June, 2007. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter 4

Rotating leader design for multi-site systems

The remainder chapters of this dissertation focuses on state machine replication for multi-site systems – a kind of {W×W} system. This chapter lays the ground work for designing efficient replicated state machine protocols for such multi-site systems by introducing the general design principle of rotating leader based protocols.

Section 4.1 explains the architecture of multi-site system. Section 4.2 introduce simple consensus, a restricted form of consensus that is later used in section 4.3 to build protocol $\mathscr{R}$ – a generic rotating leader based replicated state machine protocol. Section 4.4 analyze the performance metrics of $\mathscr{R}$ and section 4.5 discusses general optimizations that can be applied to $\mathscr{R}$.

4.1 Multi-site systems

We model a multi-site system as a collection of n sites interconnected by a wide-area network. Each site consists of a server and a group of clients. These run on separate processors and communicate through a local-area network. The servers communicate with each other through the wide-area network, and they together implement a replicated state machine. Up to f servers can fail either in a fail-stop or or in a Byzantine manner depending on the desired failure model. We also allow a server to recover later. A server serves as the main portal for its local clients, *i.e.*, unless the local server is faulty,

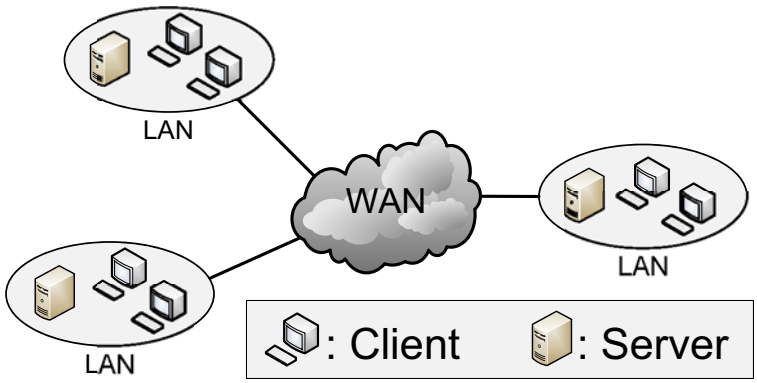


Figure 4.1: A multi-site system

clients access the replicated service by sending requests to their local server via local-area communication. When a server fails, we either assume it is acceptable for clients in the same site as a faulty server to stop making progress or for the clients to forward their requests to some remote site. Communications in the multi-site system, both within and between sites, is asynchronous, since the servers communicate through wide-area networks and the servers and clients run on generic modern operating system.

In term of network characteristics, we assume 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 (sites). The bandwidth between pairs of servers can be asymmetric and can vary over time. We do not 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 servers. This assumption holds when servers are hosted by data centers and links between centers are dedicated. It is preferable to design protocols to be adaptable to link behaviors.

One interesting application for such multi-site systems is to provide highly available services that do not suffer from co-location failures, since no two servers are in the same site. Another application is to coordinate wide-area applications that are spread across multiple sites, for example, a multi-site resource management/locking system.

Running a single leader based protocol in a multi-site system has a few disad-

vantages. For example, doing so gives performance advantages to the clients co-located with the leader over other clients. This is because in a single leader based design, all client requests are assigned to consensus instances by the leader. The would-be wide-area communication between the clients and the leader becomes local communication for clients co-located with the leader. For a system that assume Byzantine failures, such a performance advantage gives incentive for selfish servers to compete for the leader role, which in turn may lead to sub-optimal performance for the system as a whole. Even for crash failure systems, such performance advantage is unfair to the non-leader sites – it is preferable for all the sites to be able to enjoy such performance advantage, if possible. The leader may also become a performance bottleneck of the system, since the leader typically sends, receives and processes more messages than non-leader servers.

To deal with such problems, we use a *rotating leader* design. The idea is to partition the consensus sequence space so that each server is responsible for an unbounded number of instances. For example, in a system with three servers, server p_0 could be assigned the consensus instances 0, 3, 6, ..., server p_1 the consensus instances 1, 4, 7, ... and server p_2 the consensus instances 2, 5, 8 ... This design not only allows the system to avoid a performance bottleneck but also allows all the sites to share the advantage of having the leader. However, the rotating leader design only makes sense if it has no or little overhead compared to (static) leader based protocols. To achieve this we introduces *simple consensus* in the next section and use it to implement efficient rotating leader based replicated state machine protocols.

4.2 Simple Consensus

A traditional replicated state machine runs an unbounded number of instances of consensus. By design, consensus allow different servers to propose different values to the same instance of consensus, which can cause contention. To resolve contention during normal execution, a single leader is responsible for assigning requests to consensus instances. Other servers serve as backup and only take actions when they suspect the leader has failed. We take this concept one step further by introducing simple consensus, which is a restricted form of consensus.

Simple consensus is consensus with a restricted initial state. Let *no-op* to be the special state machine command that leaves the state unchanged and generates no response. In simple consensus instance i, only one distinguished server c, which we call the *coordinator* or the *owner*, can propose any command (including *no-op*); the others, which we call *followers* can only propose *no-op*. Typically, the coordinator is the default leader for the simple consensus instance.

The simplest form of simple consensus is the one in which the coordinator can propose either 0 (*no-op*) or 1 (a value other than *no-op*) and the followers can only propose 0. It is straightforward to see that the two form of simple consensus are equivalent, because the coordinator of the more complex form is bound to propose one specific value even if it has many values to chose from. We can easily map that value to 1 and *no-op* to 0.

More formally, the simple consensus problem is defined as follows: One server assume the role of coordinator; all the servers agree ahead of time on the identity of the coordinator. Each server in a simple consensus starts with a initial value to propose. The coordinator starts with either 0 or 1, and the followers start with 0. They later decides on some proposed value. A simple consensus implementation satisfies the following five properties.

Definition SC.1 (Termination): Every correct server eventually decides on some value.

Definition SC.2 (Validity): If all servers propose the same value v, then every correct server that decides decides on v.

Definition SC.3 (Integrity): Every correct server decides on at most one value.

Definition SC.4 (Agreement): If a correct server decides on v, then every correct server decides on v.

Definition SC.5 (Non-triviality): In every equivalence class relation $\equiv$ for the failure detector output, there is a run in which the correct servers decide on 1.

Note that the first four properties (SC.1 – SC.4) is the same as the four properties of consensus (C.1 – C.4). The equivalence class relation $\equiv$ for the failure detector was introduced in [16] to define WC.1 (Non-triviality) for weak consensus (see below).

Here, we use it for preventing a trivial implementation of simple consensus that decides on 0 regardless of the initial values, or merely decide on a value using the failure detector as the sole input. Note that there are only two initial state for simple consensus: (1) the coordinator proposes 0 and (2) the coordinator proposes 1. Note that, if the coordinator propose 0, the consensus outcome should be 0 according to the Validity property. So, for Non-triviality, a run that decides on 1 can occur only if the coordinator proposes 1.

It is not difficult to see that simple consensus is a restricted form of consensus and any protocol that solves consensus can be used to solve simple consensus.

Lemma 4.1: Let $\mathscr{P}$ be a protocol that implements consensus, $\mathscr{P}$ implement simple consensus.

Proof. Consensus protocol $\mathscr{P}$ can be easily transformed to a simple consensus protocol $\mathscr{P}'$ by running $\mathscr{P}$ with the simple consensus input. In additional, $\mathscr{P}'$ decides on the same value as $\mathscr{P}$.

SC.1 – SC.4: These are trivially satisfied due to C.1 – C.4.

SC.5 (Non-triviality): If the coordinator proposes 0, the outcome of $\mathscr{P}$ is 0 because all servers proposed 0. If the coordinator proposes 1, there exists a run of $\mathscr{P}$ such that 1 is decided because the initial state is bi-valent.

□

Simple consensus, however, is not simpler than consensus in term of the underlying failure detector required.

To explain, we use the *weak consensus* problem. Weak consensus is the same as consensus, except that, the Validity property is replaced by the following Non-triviality property. Assuming the only possible input values to weak consensus are 0 and 1.

Definition WC.1 (Non-triviality): In every equivalence class relation $\equiv$ for the failure detector output, there is a run in which all correct servers decide on 0 and a run in which all corrects server decide on 1.

It has been shown that even weak consensus is not deterministically solvable in asynchronous environment [26]. It is also been shown that if a failure detector $\mathscr{D}$ can

be used to solve weak consensus, it can be used to implement Ω. This makes weak consensus equivalent to consensus in term of the failure detector class required.

Lemma 4.2: If a failure detector $\mathscr{D}$ can be used to solve weak consensus, it can be used to solve simple consensus.

Proof. If $\mathscr{D}$ can be used to solve weak consensus, it can be used to implement Ω, which in turn can be used to solve consensus, which according to Lemma 4.1 can be used to solve simple consensus. □

Lemma 4.3: If a failure detector $\mathscr{D}$ can be used to implement simple consensus using protocol $\mathscr{P}$, $\mathscr{D}$ can be used to implement weak consensus

Proof. We construct a weak consensus protocol $\mathscr{P}'$ in the following way. Let v be the consensus input value of the coordinator. With $\mathscr{P}$, the coordinator proposes 1. Let the coordinator attach v with its proposal. If the simple consensus outcome of $\mathscr{P}$ is 1 then $\mathscr{P}'$ decides on v, otherwise $\mathscr{P}'$ decides on 0. $\mathscr{P}'$ solves weak consensus because:

C.1 (Termination): The termination property of simple consensus guarantees that $\mathscr{P}$ eventually terminates. If the outcome is 0, all servers eventually decides on 0. If the outcome is 1, since v is attached to the proposal 1, all servers that have decided know v already. At this point, they can decide on v.

C.3 (Integrity) and C.4 (Agreement): Trivially verified.

WC.1 (Non-triviality): In every equivalence class of relation $\equiv$ for the failure detector output, there is a run in which $\mathscr{P}$ decides on 1. In this run, $\mathscr{P}'$ decides on v. If v is 1, then $\mathscr{P}'$ decides on 1. If v is 0, then $\mathscr{P}'$ decides on 0.

□

Theorem 4.4: If $\mathscr{D}$ can be used to solve simple consensus if and only if it can be used to solve consensus.

Proof. From Lemmas 4.1, 4.2 and 4.3. □

Since simple consensus is equivalent to consensus in solvability, one might ask what is the benefit of introducing it. The primary benefit of simple consensus is that it allows the servers to learn 0 quickly if the coordinator has proposed 0. This in turn enables the design of efficient replicated state machine based on a rotating leader scheme by allowing the servers to learn *no-op* (0) quickly when they confirmed that the coordinator has proposed *no-op* (0).

Here we introduced a new terminology: the *confirmed proposal* from the coordinator. Roughly speaking, a server only confirms that a value v is proposed by the coordinator if and only if it knows that no value other than v can be confirmed at the correct servers as the value proposed by the coordinator.

Definition 4.1 (Confirmed proposal): A value v is confirmed at a correct server as the proposal by the coordinator if and only if no value other than v can be confirmed at any of the correct servers.

Lemma 4.5: In simple consensus, a server learns 0 immediately once it confirms that the coordinator has proposed 0.

Proof. By the definition of confirmed proposal, no value other 0 can be proposed by the coordinator. The followers by definition can only propose 0. Since all server can only propose 0, 0 is the only possible outcome. □

Note that confirming the coordinator has proposed a value v has different meaning under different failure models. With crash failure, as long as a server receives v as a proposal from the coordinator, a follower can confirm the coordinator has proposed v. This takes only one communication step from the coordinator to the followers. However, with Byzantine failures, a follower can not simply do so on the first proposal it receives from the coordinator, since the coordinator may tell some followers that it has proposed 0 and other followers that it has proposed 1. In this case, either 0 or 1 may become the simple consensus outcome. To deal with this problem, additional communication steps (*e.g.*, a flooding sub-protocol described in [60]) may be needed to confirm that the coordinator has proposed value v. We will revisit this when considering the Byzantine model in chapter 6.

4.3 $\mathscr{R}$: a rotating leader protocol

In this section, we explain the general design for rotating leader based replicated state machine protocols that run an unbounded sequence of simple consensus.

The goal of our design is to allow multiple servers to propose values concurrently without introducing collisions during a period of stability, which is a period during which there are no failures or false suspicions. The general idea is to run an unbounded sequence of simple consensus instances instead of consensus instances and partition the simple consensus space to allow the servers to take turns to be the coordinator and propose commands. Since each server always has a future turn to be the coordinator and propose a command, there is no need for the servers to contend, *i.e.*, if a server p is acting as the follower for an instance of simple consensus, there is no need for p to propose *no-op* when the coordinator has not been suspected. The result is a rotating leader scheme that has no contention during a period of stability. More formally, the rotating leader scheme is defined as follows:

Definition 4.2 (Rotating leader scheme): An unbounded number of simple consensus instances is run to implement a replicated state machine. For each instance, one server is designated as the coordinator, *i.e.*, the default leader of the instance. Let coordinator(i) be the coordinator of instance i. The scheme require that: (1) The assignment scheme of instances to coordinators is known by all servers; (2) $\forall n \in \mathbb{N}_0$ and for every server $p \in \{0,\ldots,n-1\}$, $\exists i > n$ such that coordinator$(i) = p$; (3) for every server $p \in \{0,\ldots,n-1\}$ there is a bounded number of instances assigned to other servers between consecutive instances that p coordinates.

The first property says there exists an agreed upon and well known assignment of coordinators. The second property restricts the assignment such that all servers always have their turn to be the coordinator. Finally, the last property further restrict this assignment to bound the "waiting time" of a server between turns. A simple round-robin scheme is an example that satisfies the above definition. This 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 dissertation.

4.3.1 Coordinated Protocols

We have shown that any consensus protocol can be used to solve simple consensus (Lemma 4.1). The benefits of the rotating leader scheme, however, lies in the use of an efficient implementation of simple consensus (Lemma 4.5). To illustrate, we introduce a class of protocols that we call *coordinated protocols* for simple consensus. In such a protocol, a server may take one of the following actions based on its role in simple consensus.

Suggest: The coordinator *suggests* a request $v \neq$ *no-op* by proposing it to other servers.

Skip: The coordinator *skips* its turn by proposing *no-op* to other servers.

Revoke: A follower *revokes* the coordinator by proposing *no-op* to other servers. Revocation is usually a result of the coordinator being suspected by a follower.

The actions *suggest*, *skip* and *revoke* specialize mechanisms that already exist in simple consensus. Making them explicit, however, enables more efficient implementation in wide-area networks. Examples of coordinated protocols include Coordinated Paxos (see section 5.3.2) for crash failure and Coordinated Byzantine Paxos (see section 6.4 for Byzantine failure. Common optimizations that can be applied to coordinated protocols include:

Default leader: The identity of the coordinator is well-known. So, the coordinator can serves as the default leader of the simple consensus instance. This way, the protocol can start from a state in which the leader status of the coordinator has be established. The coordinator can then suggest a request immediately after receiving it from the client. Doing so results in minimal latency for reaching consensus in the absence of failure and false suspicion. For example, assuming crash failure, Coordinated Paxos implements simple consensus and allows the coordinator to learn the value it suggested in just one round-trip delay, which is the theoretical lower bound.

Fast skipping: We have already explained in Lemma 4.5 that all servers can learn *no-op* as soon as it confirms that the coordinator has skipped. This allows skips to

be learned even faster than suggestions. For example, Coordinated Paxos implements skips in just one one-way communication delay from the coordinator to the follower.

4.3.2 A generic rotating leader protocol

Assume the existence of a coordinated simple consensus protocol $\mathscr{A}$. We use $\mathscr{A}$ to construct an intermediate protocol $\mathscr{R}$ for implementing replicated state machine. Note that $\mathscr{R}$ captures the essential structure of rotating leader protocols without specifying the environment in which $\mathscr{R}$ runs. By replacing $\mathscr{A}$ with appropriate protocol, we can reuse this structure to build different protocols for different environments. Note that $\mathscr{R}$ is not an efficient protocol by itself: we introduce optimizations to improve its efficiency. For example, in chapter 5, we introduce optimizations specific to the crash failure model and build Mencius. Later in chapter 6, we take similar approach for the Byzantine faulty model and we build protocol RAM. Should we need to implement or optimize replicated state machines under different assumptions, we can reuse $\mathscr{R}$ by selecting an appropriate protocol $\mathscr{A}$ and applying appropriate optimizations.

At a high level, $\mathscr{R}$ runs an unbounded sequence of simple consensus in the scheme defined by Definition 4.2. $\mathscr{R}$ commits the value learned in instance i if only if $\mathscr{R}$ has learned and committed all values learned in instances prior to i. $\mathscr{R}$ needs to handle duplicate requests that arise from, for example, the clients sending 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. Without loss of generality, we assume, for the rest of this dissertation, the latter is used and all duplicate requests are silently dropped

Each instance of simple consensus is solved using a coordinated protocol $\mathscr{A}$. We describe $\mathscr{R}$ using a set of rules that determine the behavior of a server and argue that $\mathscr{R}$ is correct using these rules.

Lemma 4.6: Protocol $\mathscr{R}$ satisfies R.2 (Agreement), R.3 (Integrity) and R.4 (Total Order).

Proof. We prove this lemma by arguing that each of the three properties holds.

R.2 (Agreement): If request r is committed by a correct server p, then the following holds: (1) r must have been decided by p in some simple consensus instance i; (2) p must have learned all instances prior to i; and (3) p does not learn r in any instance smaller than i. For any given correct server q, the Termination property of simple consensus guarantees that q will eventually learn all instances smaller than or equal to i as well. According to the Agreement property of simple consensus, the value learned in instance i is r and no value learned in instances smaller than i is r. Therefore q commit r once all instances smaller than or equal to i are learned and committed by q, which happens eventually.

R.3 (Integrity): As we have discussed, duplicate requests are handled by recording committed requests and discarding for duplicates before committing. Now, suppose a request r is committed by a correct server. Then, r must have been chosen in some simple consensus instance. The Validity property of simple consensus guarantees that r was proposed by some server in that instance, *i.e.*, previously submitted by some server.

R.4 (Total Order): If a correct server p commits r_1 before r_2, then there must exist i_1 and i_2 ($i_1 < i_2$) such that (1) p has learned all instances smaller than or equal to i_2; (2) p learns r_1 in instance i_1 and does not learn r_1 in instances smaller than i_1; and (3) p learns r_2 in instance i_2 and does not learn r_2 in instances smaller than i_2. If a correct server q commits r_2 before r_1, then there must exist j_1 and j_2 ($j_1 < j_2$) such that (1) q has learned all instances smaller than or equal to j_2; (2) p learns r_2 in instance j_1 and does not learn r_2 in instances smaller than j_1; and (3) q learns r_1 in instance j_2 and does not learn r_1 in instances smaller than j_2. Without loss of generality, we assume $i_2 \leq j_2$. Since p learns r_1 in instance i_1, according to the Agreement property of simple consensus, q must also learn r_1 in instance i_1. Since $i_1 < i_2 \leq j_2$, this contradicts with q does not learn r_1 in any instance smaller than j_2.

□

For R.1 (Validity), we use the following Rules RL.1 – RL.4 to ensure that any client requests sent to a correct server eventually commit.

As we have discussed, one of the benefits of using the rotating leader design is the ability to minimize the commit latency. To do that we let a server suggest a value immediately upon receiving it from a client.

Rule RL.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 coordinates, *i.e.*, $I'_p = \min\{k : \text{coordinator}(k) = p \wedge k > i\}$, where I'_p is the updated index.

Rule RL.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 RL.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 RL.2 uses a technique similar to logical clocks [35] to overcome this problem.

Rule RL.2: If server p receives a suggestion for instance i and $i > I_p$, before proceeding with protocol $\mathscr{A}$, p updates I_p such that its new index $I'_p = \min\{k : \text{coordinator}(k) = p \wedge k > i\}$. p also executes skip actions for each of the instances in the set $\{i : I_p \leq i < I'_p \wedge \text{coordinator}(i) = p\}$, *i.e.*, all instances in the range $[I_p, I'_p)$ that p coordinates.

With Rule RL.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 faulty server may not skip, for example, because it is crashed or because it is acting in a Byzantine manner. Such a server can prevent others from committing. Rule RL.3 overcomes this problem.

Rule RL.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.

Rule RL.3 allows the non-faulty servers to fill in the gaps left by faulty servers with *no-op* so that the correct servers can make progress. In principle, all non-faulty

servers can start revocation upon the suspicion of another server. With the crash failure model, this means any non-faulty server can start revocation against a suspected server at any time. In practice, however, one non-faulty server is usually elected to lead the revocation process to avoid wasting resources or causing liveness problems (see section 5.4.3). With the Byzantine failure model, in addition to electing a leader to lead the revocation process, at least $f+1$ suspicions against the same server are required to start revocation: since up to f servers can fail in a Byzantine manner, $f+1$ suspicions ensure that at least one of them is from a non-faulty server.

Lemma 4.7: When there is no false suspicion, $\mathscr{R}$ with Rule RL.1 – RL.3 satisfies R.1 (Validity).

Proof. 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 suggestion. Thus, according to Rule RL.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 RL.3, non-faulty servers eventually revoke them, and non-faulty servers eventually learn the outcome. When instances prior to i are eventually learned and committed, the value learned in instance i can then be committed. Since we assume there is no false suspicion, the followers do not take the revocation action. So, v, the value suggested by the coordinator, is eventually chosen as the outcome of instance i and eventually committed. □

False suspicions, however, are possible with unreliable failure detectors and can potentially occur an unbounded number of times. We add Rule RL.4 to allow a server to suggest a request multiple times upon false suspicions.

Rule RL.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.

Ω has been shown to be the weakest failure detectors to solve consensus, and $\Diamond\mathbb{W}$ and Ω are equivalent classes of failure detectors [16]. However, they are not sufficient to implement a replicated state machine with $\mathscr{R}$. For example, $\Diamond\mathbb{W}$ only guarantees

there is a time after which at least one correct server is never suspected. In other words, some correct server could be permanently falsely suspected and hence be revoked for unbounded number of instances in $\mathscr{R}$, keeping this server and its clients from making progress. There are two ways for dealing with this problem. One way is to use a stronger failure detector, for example $\Diamond\mathbb{P}$ – the next stronger commonly-used failure detector than $\Diamond\mathbb{W}$.

Lemma 4.8: Assuming $\Diamond\mathbb{P}$, $\mathscr{R}$ with Rule RL.1 – RL.4 satisfies RSM-Validity.

Proof. If any correct server p suggests a value v. By Rule RL.4, $\mathscr{R}$ will continue to re-suggest v upon false suspicion until v is chosen. By the definition of $\Diamond\mathbb{P}$, there exist a time t after which p will not be suspected. If v hasn't been chosen by t , by Lemma 4.7, v will be chosen once p re-suggested v after t. Therefore, $\mathscr{R}$ satisfies RSM-Validity. □

Theorem 4.9: Assuming $\Diamond\mathbb{P}$, $\mathscr{R}$ with Rule RL.1 – RL.4 implements replicated state machines.

Proof. From Lemma 4.6 and Lemma 4.8. □

Note that $\Diamond\mathbb{P}$ is strictly stronger than either $\Diamond\mathbb{W}$ and Ω. In practice, when assuming $\Diamond\mathbb{P}$ a period of no false suspicion only needs to hold long enough for p to re-suggest v and have it chosen for the protocol to make progress. This is not much more difficult to implement than $\Diamond\mathbb{W}$ or Ω under the crash failure model. It is, however, considerable more difficult when the Byzantine failure model is considered: for example, faulty servers can trick the correct servers to consider a correct server is being maliciously slow when the slowness is caused by high network latency. When using $\Diamond\mathbb{P}$ is impossible, another way to deal with the insufficiency of $\Diamond\mathbb{W}$ or Ω is to allow a server to forward its request to another server when it is repeated falsely suspected. This way, a server is able to make progress as long as it can find a correct server to forward its request to. A server can eventually do so since we assume the existence of Ω failure detector – the weakest failure detector for solving asynchronous consensus. Note that, forwarding is a degeneration from the rotating leader design to the single leader design. In the extreme case, if all but one correct server are being falsely suspected, all other servers forward their requests to that server: it becomes the same as having a single server to propose

the request for all servers. So, the single leader design can be viewed as a special case of the rotating leader design with the forwarding mechanism. We formalize the forwarding mechanism using Rule RL.5 and prove its correctness with Theorem 4.10.

Rule RL.5: If server p suggests a value $v \neq$ *no-op* to instance i, and p learns that *no-op* is chosen, instead of re-suggesting v, p may opt to forward v to another server r that p believes is correct. Upon receiving v, r treats it the same as a client request.

Rule RL.5 implies that if p later suspect r has failed and has not learned v yet, it may forward v to another server r'.

Theorem 4.10: Assuming Ω, $\mathscr{R}$ with Rule RL.1 – RL.5 implements replicated state machines correctly.

Proof. From Lemma 4.6, we know that $\mathscr{R}$ always satisfies R.2 (Agreement), R.3 Integrity and R.4 (Total Order). So, we only need to prove R.1 (Validity). From Lemma 4.7, we know that only the requests sent to falsely suspected servers are not live. Rule RL.5 allow a falsely suspected server p to forward request to another server r and r will suggest the request. p can eventually find a correct server r to forward to request to because we assume Ω. Since any request suggested by a correct server is guaranteed to be live, any request submitted to p is also guaranteed to be live. So, $\mathscr{R}$ with Rule RL.1 – RL.5 satisfies Validity and hence implements replicated state machines correctly. □

4.4 Performance analysis of $\mathscr{R}$

Table 4.1: During a period of stability, the performance metrics of coordinated simple consensus protocol $\mathscr{A}$ derived from consensus protocol $\mathscr{C}$

Protocol type		Simple consensus ($\mathscr{A}$)			Consensus ($\mathscr{C}$)
Action		Suggest	Skip	Revoke	Propose
Learning latency	Leader/Coordinator	L_{sc}	0	L_r	L_c
	follower	L_{sf}	L_k		L_f
Message complexity		N_s	N_k	N_r	N
Quorum size		Q_s	Q_k	Q_r	Q

Key metrics, such as number of message delays, message complexity and quorum size can be helpful for predicting the performance of a protocol. In this section, we analyze these metrics for $\mathscr{R}$. Since $\mathscr{R}$ is built by running protocol $\mathscr{A}$ repeatedly, these metrics, without any optimization, are mostly determined by the underlying protocol $\mathscr{A}$ that is used. We only analyze these metrics for the common case, *i.e.*, during a period of stability. This is because during unstable periods, during which there exist new failures or false suspicions, no protocol can guarantee liveness – it may take arbitrary long communication and arbitrary large number of messages to reach consensus. We also assume a consensus protocol $\mathscr{C}$ is used to derive a coordinated simple consensus protocol $\mathscr{A}$, which in turn is use to built $\mathscr{R}$.

Table 4.1 summarizes the performance metrics of $\mathscr{A}$ and $\mathscr{C}$. In the table, we use L to denote learning latency, N to denote message complexity, and Q to denote quorum size. We also use subscript s to denote the suggestion action, subscript k to denote the skip action, subscript r to denote the revoke action, subscript c to denote the coordinator or the leader, and subscript f to denote the followers.

The following relationship between the metrics of $\mathscr{A}$ and $\mathscr{R}$ can be assumed:

- $L_k < L_{sc} \leq L_{sf} < L_r$, $N_k < N_s < N_r$, and $Q_k < Q_s$: $L_k < L_{sc}$, $N_k < N_s$, and $Q_k \leq Q_s$ because skipping is a simpler operation than suggesting according to Lemma 4.5. $L_{sc} \leq L_{sf}$ because the leader always learns no slower than the followers and the leader can potentially learn faster in some cases due to certain optimizations. $L_{sf} < L_r$ and $N_s < N_r$ because the revocation process involves leader change, which results in higher latency and larger message complexity.

- $L_{sc} \leq L_c < L_r$, $L_{sf} \leq L_f < L_r$, and $N_s \leq N$: $L_{sc} \leq L_c$, $L_{sf} \leq L_f$, and $N_s \leq N$ because suggesting in the derived protocol $\mathscr{A}$ typically follows the same execution path as that of leader proposing a value, though some simple consensus specific optimization may potentially reduce the communication steps for $\mathscr{A}$. $L_c < L_r$ and $L_f < L_r$ because revocation, which typically involves leader change, requires longer communication chain than the failure-free proposing action of $\mathscr{C}$.

- $Q_s = Q_r = Q$. This is because the derived protocol $\mathscr{A}$ still needs the same number of replicas as $\mathscr{C}$ to make progress.

Table 4.2: During a period of stability, the performance metrics of state machine $\mathscr{R}$ compared to consensus protocol $\mathscr{C}$.

			State machine $\mathscr{R}$	Relation	$\mathscr{C}$
No failure	Commit latency	Leader	$\max\{L_{sc}, L_k+1\}$	$\leq$	L_c
		Follower	$\max\{L_{sf}, L_k+1\}$	$\leq$	L_f
	Message complexity		$N_s+(n-1)N_k$	$>$	N
	Quorum size		n	$>$	Q
f failures	Commit latency	Leader	$\max\{L_{sc}, L_k+1, L_r\}$	$>$	L_c
		Follower	$\max\{L_{sf}, L_k+1, L_r\}$	$>$	L_f
	Message complexity		$N_s+(n-f-1)N_k+fN_r$	$>$	N
	Quorum size		$n-f$	$\geq$	Q

Using the values in Table 4.1, Table 4.2 summarizes the performance metrics of $\mathscr{R}$ and compares them to those of $\mathscr{C}$. Note that concurrent requests are not considered in Table 4.2. We offer the following explanations for Table 4.2:

Learning latency (no failures): When no server has failed, the commit latency for $\mathscr{R}$ at the coordinator p is the larger of L_{sc} and L_k+1. This is because even it only takes L_{sc} steps for the coordinator p to learn the value it suggested, p can not commit the value until it knows that the other servers have skipped their turns. This takes L_k+1 steps, since the other servers starts skipping immediately after they receives the suggestion from p and it takes additional L_k steps for p to learn the skips. Assuming $\mathscr{A}$ is skip-fast, we know $L_k < L_{sc}$. And so, $\max\{L_{sc}, L_k+1\} = L_{sc} \leq L_c$. Similar result can be obtained for the followers, as shown in Figure 4.2.

Learning latency (f servers failed): Assume f servers have failed and all the other servers know which are the ones that have failed. When a correct server suggests a value, it also needs to revoke the faulty ones and wait for the result of the revocation before committing its request. So, the learning latency at the coordinator is $\max\{L_{sc}, L_k+1, L_r\}$. We already know that $L_k < L_{sc} < L_r$, and $L_r > L_c$ so $\max\{L_{sc}, L_k+1, L_r\} = L_r > L_c$. Similar results can also be obtained for the followers, as shown in Figure 4.2.

Message complexity: When concurrent requests are not considered, n simple consen-

sus instances are consumed for committing one request: one to choose the request, f to revoke the f known faulty servers, and $n-f-1$ are skipped. So, the message complexity of $\mathscr{R}$ is $N_s+(n-f-1)N_k+fN_r$, which is considerably higher than N, considering that $N_r > N$. When there is no failures, *i.e.*, $f = 0$, the message complexity is $N_s+(n-1)N_k$, which is still higher than N, considering that usually $N_s = N$ and $N_k < N$.

Quorum size: For protocol $\mathscr{A}$, Q_s and Q_r, the quorum size of $\mathscr{A}$ for suggestion and revocation, are usually the same as that of protocol $\mathscr{C}$, which is Q. Q_k, the quorum size for skipping, is no larger than Q because skipping can be optimized in $\mathscr{A}$, but the coordinator is a critical server in the case of skipping. And since a server in $\mathscr{R}$ needs to wait for all non-faulty servers to skip, it needs to wait for $n-f$ servers (including itself) before committing a request.

We have not yet considered concurrent requests. When requests are proposed concurrently by servers, fewer simple consensus instances need to be skipped or revoked. So, concurrent requests can help cut down the message complexity of $\mathscr{R}$. Assuming m requests are concurrently proposed for n consecutive simple consensus instances that coordinated by n distinct servers, the amortized message complexity for committing one request is $(mN_s+(n-f-m)N_k+fN_r)/m$, which is still higher than N, unless $m = n$ and $f = 0$, *i.e.*, there is no failure and all servers propose requests at the same time.

From the above analysis, we know concurrent requests reduce message complexity. However, concurrent requests also introduces contention, which can cause higher commit latency for $\mathscr{R}$. We analyze this case in the next subsection.

4.4.1 Delayed commit

While any server of $\mathscr{R}$ can commit its request in just L_{sc} steps when there is no contention, commits may have to be delayed up for additional steps when there are concurrent suggestions.

For example, in the scenario illustrated in Figure 4.2, server p_0 suggests x to instance 0 concurrently with p_1 suggesting y to instance 1. L_{sc} steps after proposing

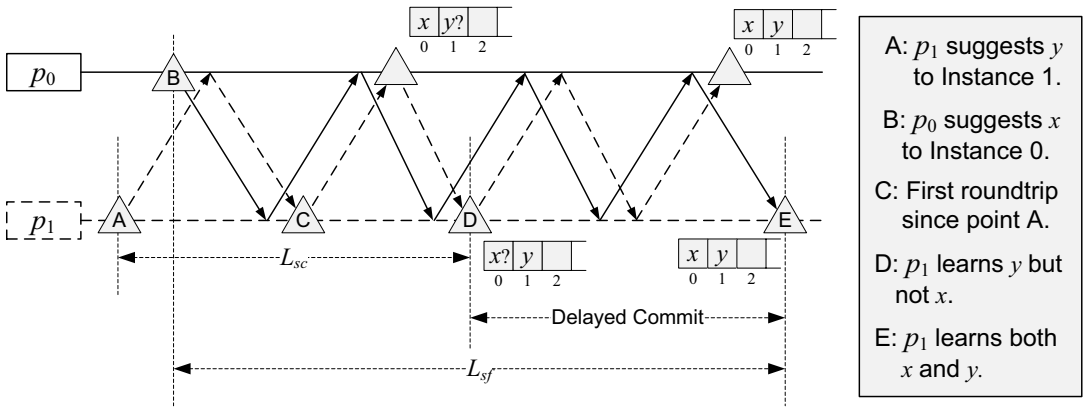


Figure 4.2: Delayed commit

y, p_1 learns that y has been chosen for instance 1, but cannot commit y yet because instance 0 is still in progress and a value is yet to be learned for instance 0. In this case, p_1 cannot commit y until L_{sf} steps after p_0 suggested x. Then p_1 can commit both x and y at once. We say that y experiences a *delayed commit* at p_1. The extra delay introduced by delayed commit is bounded by Theorem 4.11. We also discuss a simple flexible commit mechanism in section 4.5.2 to mitigates the effect of delayed commit.

Theorem 4.11: Assume (1) the coordinator and the followers learns the suggestion from the coordinator in L_{sc} and L_{sf} steps respectively; (2) the servers skip unused turns immediately after receiving the suggestion from the coordinator; and (3) FIFO communication channels. Let L_d be the extra delay caused by delayed commit, then $L_{sf} - L_{sc} - 1 < L_d < L_{sf} - L_{sc} + 1$.

Proof. Without losing generality, we use the example in Figure 4.2. In the example, p_0 and p_1 suggest x and y to instances 0 and 1 respectively.

For the delayed commit to occur at p_1, p_1 must receives the suggestion of x in between point A and C, *i.e.*, after p_1 suggests y and before p_1 receives the first round-trip from p_0 after the suggestion. The suggestion of x must be received after the p_1 suggests y because p_1 would have skipped instance 1 and no delayed commit would happen. Suppose the suggestion of x is received after the first round trip, *i.e.*, point C. We know p_0 must have suggested x (point B) after receiving the suggestion of y from p_1 due to the

FIFO channels. This would have made p_0 skip instance 0, avoiding the delayed commit. So, the suggestion must be received before point B.

When delayed commit occurs, L_y, the total delay of y, is from point A to E. From the above argument we know that $L_{sf} - 1 < L_y < L_{sf} + 1$: when the suggestion of x is received right after point A at p_1, L_y is minimized to $L_{sf} - 1$ steps; when the the suggestion of x is received right before point c at p_1, L_y is maximized to $L_{sf} + 1$. Since it takes L_{sc} steps for y to be learned at p_1, we have $L_{sf} - L_{sc} - 1 < L_d < L_{sf} - L_{sc} + 1$.

□

4.5 Optimizations for $\mathscr{R}$

From the performance analysis in the previous section, it is straightforward to see that protocol $\mathscr{R}$, without optimizations, offers little advantage over $\mathscr{C}$. The only case $\mathscr{R}$ is better than $\mathscr{C}$ is when there are no concurrent requests, $\mathscr{R}$ allows all servers to be able to commit their requests in L_c steps, which is only possible at the leader with $\mathscr{C}$. However, concurrent requests can make this advantage go away, since delayed commit adds extra latency to the baseline latency. Other drawback of $\mathscr{R}$ includes: high message complexity, increased latency when one or more server has failed due to revocation, and the need to receive skips from all correct servers to make progress. It is obvious that optimizations are necessary for any protocol based on $\mathscr{R}$ to be practical. In this section, we discuss optimizations that are generic to protocol $\mathscr{R}$, *i.e.*, those that do not involve the specific detail of the simple consensus protocol $\mathscr{A}$ used. These include issuing revocation in large block size and the use of *out-of-order commit*, a flexible commit scheme that is unique to rotating leader based protocols because of the properties of simple consensus. There are also other forms of optimizations that can be applied to rotating leader based protocols, however, they do involves the details of protocol $\mathscr{A}$, and so we discuss them later when we introduce the specific protocols for solving replicated state machine in specific environments.

4.5.1 Revocation in large blocks

Revocation was introduced in protocol $\mathscr{R}$ to ensure liveness in the presence of faulty servers. Rule RL.3 allows a server p to revoke instances coordinated by faulty servers prior to I_p, *i.e.*, the index of server p. This introduces a performance bottleneck when one or more servers have failed: revocation, which has high communication delays and large message complexity, is executed regularly in the normal execution path. A simple idea is to revoke all q's future turns, which irreversibly chooses *no-op* for all q's future turns. However, q may need to suggest values in the future, either because q was falsely suspected or because it recovers. A better way to reduce the cost of revocation is to revoke faulty servers in advance of the index of p and in large block as described in Optimization RL.1.

Optimization RL.1: 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 RL.1 allows p to revoke q at least β instances in advance before p suggests a value to some instance i greater than C_q. The rationale behind Optimization RL.1 is that a faulty server should be revoked in the future anyway, revoking it in advance removes revocation from execution path of normal proposing activities. By tuning β, we can 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 also reduced.

Lemma 4.12: Assuming $\Diamond\mathbb{P}$, protocol $\mathscr{R}$ (RL.1 – RL.4) with Optimization RL.1 implements replicated state machines correctly.

Proof. Note that Optimization RL.1 can only exclude the actions of a falsely suspected server for a bounded number of instances. It clearly does not violate any of the safety properties. Assuming $\Diamond\mathbb{P}$ means that such false suspicions will eventually cease. So, using Optimization RL.1 does not affect the liveness of the protocol. □

In chapter 5, we use Lemma 4.12 to prove the correctness of protocol Mencius. Similarly, Lemma 4.13 can be proved for forwarding based protocol. We use Lemma 4.13 to prove the correctness of protocol RAM in chapter 6.

Lemma 4.13: Assuming Ω, protocol $\mathscr{R}$ (RL.1 – RL.5) with Optimization RL.1 implements replicated state machines correctly.

Proof. Note that Optimization RL.1 can only exclude the actions of a falsely suspected server for a bounded number of instances. It clearly does not violate any of the safety properties. Assuming Ω means that such false suspicion will eventually cease for the elected leader. Since Optimization RL.1 only extend the false suspicion period of the leader for a bounded period, the resulting failure detector still satisfies the property of Ω. So, using Optimization RL.1 does not affect the liveness of the protocol. □

4.5.2 Out-of-order commit

In the example shown in Figure 4.2, 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.

Theorem 4.14: When a request y suggested to instance i and originated from server p_1 experiences a delayed commit because of a request x suggested to instance j ($j < i$) and originated from server p_0, p_1 can commit y immediately if (1) x and y are commutable; and (2) p_1 has confirmed that p_0 has suggested value x to instance j.

Proof. Once p_1 has confirmed that that x is suggested by p_0 to instance j, the outcome of instance j is narrowed down to two values x or *no-op*. This is true because p_0, the coordinator of instance j, has proposed x, all the other servers, *i.e.*, the followers, can only propose *no-op* by the definition of simple consensus. If x and y are commutable, y will be commutable with the outcome of instance j regardless which value is eventually chosen because *no-op* commutes with any requests. So, y can be committed at p_1 as soon as p_1 has confirmed that x was suggested by p_0. □

The out-of-order commit mechanism is unique to the rotating leader protocols and can not easily applied to generic consensus based protocols because out-of-order commit takes advantages of the definition of simple consensus, which restricts the value that can be proposed by the followers – the key for $\mathscr{R}$ to guarantee safety while allowing out-of-order commit.

Note that Theorem 4.14 only discusses delayed commit between two instances of simple consensus. However, it is possible for a request to have multiple dependencies caused by delayed commit. In this case, the request can not be committed until all the depended requests are committed or all uncommitted requests are confirmed to be commutable with itself.

Finally, whether two requests are commutable is application dependent. For out-of-order commit to work, this means, in additional to the propose and commit interface used by replicated state machine, another interface is need to ask the application if requests can be commuted. See section 5.5.5 and section 6.8.3 for evaluations of the effectiveness of out-of-order commit.

4.6 Summary

In this section, we introduced the simple consensus problem, which we later proved to be equivalent to consensus in term of solvability. We then explained $\mathscr{R}$: a generic rotating-leader replicated state machine protocol that runs an unbounded sequence of simple consensus instead of consensus. This makes $\mathscr{R}$ easily satisfies the Agreement, Integrity and Total Order properties of replicated state machines. The Validity of the protocol is ensured using a set of rules. Depending on the failure detectors assumed, four or five rules may be used. In the next chapter, we instantiate and optimize $\mathscr{R}$ to obtain an efficient crash-failure replicated state machine protocol that we call Mencius.

Chapter 5

Crash Failure: Protocol Mencius

In this chapter, we study efficient replicated state machine protocols for multi-site {W×W} systems under the crash failure model. Our goal is to build a crash failure protocol that has 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.

Existing protocols such as Paxos, Fast Paxos, and CoReFP [25] are not, in general, the best consensus protocols for such 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 chapter presents *Mencius*[1], a rotating-leader replicated state machine 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.

[1]Mencius (Chinese: 孟子; pinyin: Mèng Zǐ) 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.

The basic approach of Mencius is to partition the sequence of consensus protocol instances among the servers, much as protocol $\mathscr{R}$ does. Indeed, Mencius is an optimized version of $\mathscr{R}$ that uses Coordinated Paxos, a variant of Paxos, for solving simple consensus. Applying the rotating leader scheme of $\mathscr{R}$ amortizes the load of being a leader. Doing so increases throughput when the system is CPU-bound. When the network is the bottleneck, a rotating leader scheme is capable of fully utilizing the available bandwidth to increase throughput. It also reduces latency, because clients can use a local server as the leader for their requests; and because of the design of $\mathscr{R}$, 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 [40], 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. Simple consensus plays a key factor in the design of Mencius for achieve these performance improvements First, 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. Second, 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 section is as follows. Section 5.1 examines the performance problems in a crash-failure multi-site environment. Section 5.2 discuss the drawbacks of existing protocols in multi-site environment. Section 5.3 refines Paxos into Mencius. Section 5.4 discusses practical implementation considerations for Mencius. Section 5.5 evaluates Mencius, Section 5.6 summarizes related work, and Section 5.7 discusses future work and open issues. Section 5.8 summarizes this chapter.

5.1 Performance issues

In this section we brief review the performance issues and system bottleneck in a multi-site system to better understand how to design an efficient crash failure replicated

state machine protocol.

In the multi-site system, the clients access the service by issuing requests to the servers. From the clients' point of view, the most important performance metric is to have low request latency, *i.e.* low delay in time between a request is issued and a reply is received. In addition to low latency, high throughput is also desirable because the latency of the requests can increase dramatically in a low throughput system when the system is under high load: the requests that cannot be processed in time build up queues at the servers and end up spending more time in queue than being processed [28]. So, it is our goal to design a protocol with both high throughput and low latency.

For throughput, there are two possible bottlenecks in this systems, 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. Limiting the use of message broadcasting helps to reduce the message complexity.

To achieve low latency, it is important to design the protocol that takes advantage of the multi-site system architecture, *i.e.*, the communication within a site has much higher throughput and lower latency than that across sites. And so, it is important to have short chains of wide-area communication steps for the servers to learn the consensus outcome. However, the number of communication steps may not be the only factor that impacts latency: high variance on the delivery of message in wide-area networks is also a major contributor (see section 3.3).

Finally, besides high throughput and low latency, it is also preferable to design a protocol that adapts well to the network environment. For example, for inter-site communication, the link latency have high variance and the link bandwidth may also change from time to time, an adaptive protocol works well with a wide-range of such network conditions and adjusts itself when network condition changes.

5.2 Why not existing protocols?

Over the year, a large number of crash failure protocols have been proposed. Most of them, however, were primarily designed for local area networks. Not surprisingly, even the best existing protocols, such as Paxos, Fast Paxos and CoReFP, have their respective drawbacks in multi-site systems. In section 3.3, we showed that a protocol designed for {L×L} settings may not work as well in {W×L} settings. More specifically, we compared Paxos and Fast Paxos. Table 5.1 extends this comparison to multi-site {W×W} settings.

Table 5.1: A comparison of Paxos and Fast Paxos in multi-site systems

	Paxos	Fast Paxos
WAN learning latency	Leader:2 Follower: 4	2 (when no collision)
Collision	No	Yes
Number of replicas	$2f+1$	$3f+1$
WAN message complexity	Leader: $3n-3$ Follower: $3n-2$	n^2-1
Load balancing	No	Yes

Fast Paxos has the advantages in two categories: (1) When collision and contention do not occur with Fast Paxos, client requests at all sites have the minimal latency of two wide-area delays. With Paxos, two-step delay is only possible at the leader site, while all other sites have a latency of four wide-area delays. (2) In Fast Paxos, all servers act the same, and so, the protocol is more load-balanced than the leader-based Paxos. The two advantages, however, likely translate into little real-life performance advantages for Fast Paxos because (1) concurrent requests result can collide, which not only nullifies the advantage of reduced latency of Fast Paxos but also adds extra latency

required for recovery; (2) though more load-balanced, the high message complexity of Fast Paxos result in lower throughput when the system is CPU-bound: each Fast Paxos server may end up with higher load than the leader with Paxos. Finally, Fast Paxos also require more replicas than Paxos to tolerate the same number of failures. This not only increases the initial hardware cost but also increases the subsequence maintain cost: more servers means more failures.

CoReFP [25] tries to deal with the collision problem of Fast Paxos by running Paxos and Fast Paxos concurrently. When there is no contention, it has the minimal two-step latency at all servers. When concurrent requests result in contention, one more step is necessary to reach consensus. However, the system has a larger message complexity than both Paxos and Fast Paxos. As a result, it has lower throughput and a significant penalty on latency when system load is high.

Though not clear cut, Paxos 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. Paxos, however, is still not ideal for multi-site systems due to its single-leader nature. We examine this next.

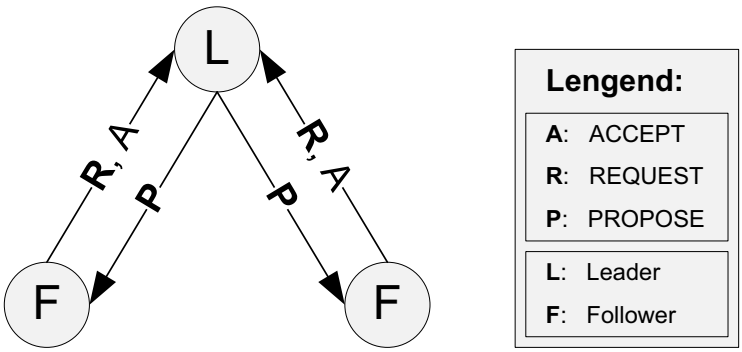


Figure 5.1: The communication pattern in Paxos

Unbalanced communication pattern: With Paxos, the leader generates and consumes more traffic than the other servers. Figure 5.1, 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 REQUEST 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, a REQUEST is sent from the originator to the leader and a PROPOSE is broadcast by the leader. Under high load, the outgoing bandwidth of the leader is the bottleneck, whereas the channels between the non-leaders are idle. In contrast, Mencius uses a rotating leader scheme. Both eliminates the need to send REQUEST messages across wide area network and 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 as n scales up. By rotating the leader, 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 REQUEST 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 [34]. 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

using out-of-order commit.

5.3 Deriving Mencius

Mencius is a rotating-leader protocol designed for crash failure multi-site systems. In this section, we first explain our assumptions and design decisions. We then introduce Coordinated Paxos, an efficient variant of Paxos for solving simple consensus. We later use Coordinated Paxos to instantiate the generic rotating-leader protocol $\mathscr{R}$. Finally, we optimize the instantiated protocol. 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 rules, optimizations, and accelerators, it is relatively easy 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 environment.

5.3.1 Assumptions

In addition to the crash failure multi-site system model, we make the following assumptions about the system. Note that it is possible to build a variant of Mencius that does not rely on these assumptions. However, these assumptions do make the derivation and analysis of the protocol cleaner.

Recoverable crash-failure process: 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. Stable storage simplifies recovery from failures that only last a short period of time. However, more sophisticated method need to be considered for recovering from longer failures (see section 5.4.2.

Unreliable failure detector $\Diamond\mathbb{P}$: We have already explained that $\mathscr{R}$ requires $\Diamond\mathbb{P}$ for liveness. So does Mencius, a protocol derived from $\mathscr{R}$. Though it is possible for Mencius to use the weaker Ω by using the forwarding mechanism described in the previous chapter, we do not consider it here.

Asynchronous FIFO communication channe: 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 made by 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 the optimizations discussed in section 5.3.4. 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 5.4.2).

5.3.2 Coordinated Paxos

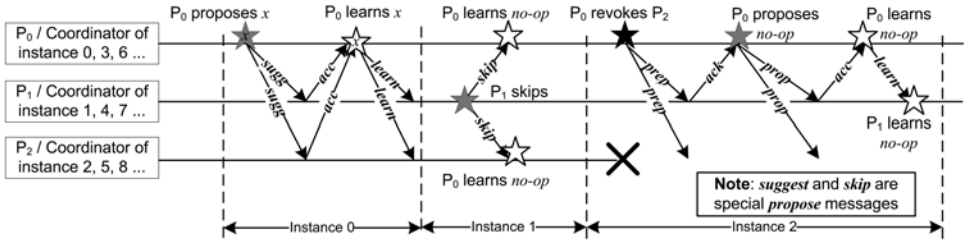


Figure 5.2: The message flow of suggest, skip and revoke in Coordinated Paxos.

It is easy to see that Paxos, a consensus protocol, implements simple consensus as a direct result of Lemma 4.1.

Lemma 5.1: Paxos implements simple consensus.

We use, however, an efficient variant of Paxos to implement simple consensus. We call it *Coordinated Paxos* (see Appendix A for the protocol in 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_0. Such a state consists of a promise not to accept any value for any round smaller than r_0. A server can subsequently initiate the following actions, as shown in Figure 5.2:

Suggest: The coordinator *suggests* a request v by sending PROPOSE messages with payload v in round r_0 (Instance 0 in Figure 5.2). We call these PROPOSE messages SUGGEST messages.

Skip: The coordinator *skips* its turn by sending PROPOSE messages that proposes *no-op* in round r_0 (Instance 1 in Figure 5.2). 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 5.2). Just like a new Paxos leader would do, it starts Phase 1 for some round r' greater than the current round 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.

As we described in chapter 4, 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. Also, it is straightforward to see that SUGGEST and SKIP are specialized PROPOSE messages that propose a client request and *no-op* to round r_0 respectively.

Theorem 5.2: Assuming a failure detector at least as strong as Ω, Coordinated Paxos implements simple consensus.

Proof. By Lemma 5.1, we know Paxos implements simple consensus. Coordinated Paxos differs from Paxos in the followings: (1) Coordinated Paxos starts from a specific (and safe) state; (2) a server learns *no-op* upon receiving a SKIP message from the coordinator, and can act accordingly; and (3) Coordinated Paxos assumes a failure detector at least as strong as that assumed by Paxos. None of the three affects Paxos's safety and liveness in term of implementing simple consensus, therefore, Coordinated Paxos implements simple consensus. □

5.3.3 $\mathscr{P}$: A simple state machine

We now construct an intermediate protocol $\mathscr{P}$ by using Coordinated Paxos to instantiate protocol $\mathscr{R}$. The pseudo code of $\mathscr{P}$ can be found in Appendix B. Essentially, $\mathscr{P}$ follows the same set of rules that $\mathscr{R}$ does (Rule RL.1 – RL.4). For the purpose of completeness, we restate these rules in the context of Coordinated Paxos.

Rule M.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 M.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 $I'_p = min\{k : p \text{ coordinates instance } k \wedge k > i\}$. p also executes skip actions for each of the instances in range $[I_p, I'_p)$ that p coordinates.

Rule M.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.

Rule M.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.

Corollary 5.3: Assuming $\Diamond\mathbb{P}$, $\mathscr{P}$ implements replicated state machines.

Proof. $\mathscr{P}$ is instance of $\mathscr{R}$ by instantiate $\mathscr{A}$ using Coordinated Paxos. So, $\mathscr{P}$ implements replicated state machines, according to Theorem 4.9. □

5.3.4 Optimizations

Protocol $\mathscr{P}$ is correct but not necessarily efficient. It always achieves the minimal two communication steps for the proposing server to learn the consensus value, but its message complexity varies depending on the rates at which the servers suggest values. In the worst case, when there is no failure and only and only one server is proposing requests, the message complexity of $\mathscr{P}$ is $(n+2)(n-1)$: $3n-3$ for suggesting the value

and $(n-1)^2$ for the rest of servers to skip. This is considerably higher than that of Paxos under the same condition ($3n-3$). In this section, we optimize $\mathscr{P}$ to obtain Mencius.

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 augment 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 M.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.

An alternative to implement Optimization M.1 is to include i_1 as an additional field in the ACCEPT message when FIFO channels are not available. This alternative can be applied to Optimization M.2 (shown below) as well.

Lemma 5.4: Protocol $\mathscr{P}$ with Optimization 1 implements replicated state machines correctly.

Proof. Optimization M.1 combines multiple messages that $\mathscr{P}$ would have sent separately into one message. Doing this clearly does not violate any of the safety properties. It does not affect the liveness properties either, since the combined message is sent with no additional delay. Therefore, Protocol $\mathscr{P}$ with Optimization M.1 is correct. □

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:

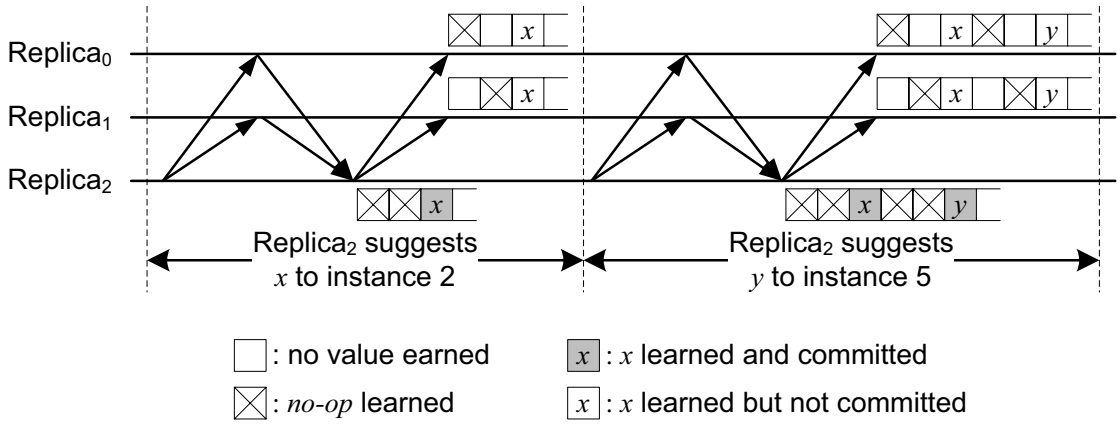


Figure 5.3: Liveness problem can occur when using Optimization M.2 without Accerlerator M.1

Optimization M.2: p does not immediately send a SKIP message to r. 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.

An alternative way to propagate the SKIP messages from p to other servers is to have p piggyback them on instance i's Phase 3 LEARN messages that are broadcast by p. However, it is less scalable than Optimization M.2 because the number of additional fields needed grows linearly with the number of server n. We do not discuss this alternative further in this dissertation.

Note that Optimization M.2 can potentially defer the propagation of SKIP messages from p to r for an unbounded period of time. Figure 5.3 shows one such 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:

Accerlerator M.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.

Lemma 5.5: Protocol $\mathscr{P}$ with Optimization M.2 and Accelerator M.1 implements replicated state machines correctly.

Proof. Optimization M.2 combines multiple messages that $\mathscr{P}$ would have sent separately into one message. Doing this clearly does not violate any of the safety properties. Optimization M.2 alone, however, could affect the liveness properties, since it can potentially delay the propagation of SKIP messages for an unbounded amount of time. Because Accelerator 1 bounds the delay and $\mathscr{P}$ only relies on the eventual delivery of messages for liveness, adding Optimization M.2 and Accelerator M.1 to protocol $\mathscr{P}$ does not affect liveness, and so $\mathscr{P}$ still implements replicated state machines. □

Given that the number of extra SKIP messages generated by Accelerator M.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 REQUEST 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. We have already explained how to reduce

the cost of revocation using Optimization RL.1 for $\mathscr{R}$. For the purpose of completeness, we restate it here as Optimization M.3.

Optimization M.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 $C_q < I_p + \beta$.

Corollary 5.6: Protocol $\mathscr{P}$ with Optimization M.3 implements replicated state machines correctly.

Proof. From Lemma 4.12. □

Optimization M.3 addresses the common case where there are no false suspicions. When a false suspicion does occur, it may result in poor performance for those falsely suspected servers. 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 5.4.5).

Mencius is $\mathscr{P}$ combined with Optimizations M.1 – M.3 and Accelerator M.1. Appendix C has the pseudo code for Mencius.

Theorem 5.7: Mencius implements replicated state machines correctly.

Proof. From Lemma 5.4, 5.5 and Corollary 5.6. □

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. Figure 5.4 compares the communication pattern in Paxos and Mencius. The links between Paxos followers are left idle whereas they are utilized in Mencius. By distrbuting messages that carry significant payload to all the available links, Mencius is able to achieve better throughput under network-bound payload (see section 5.5.2).

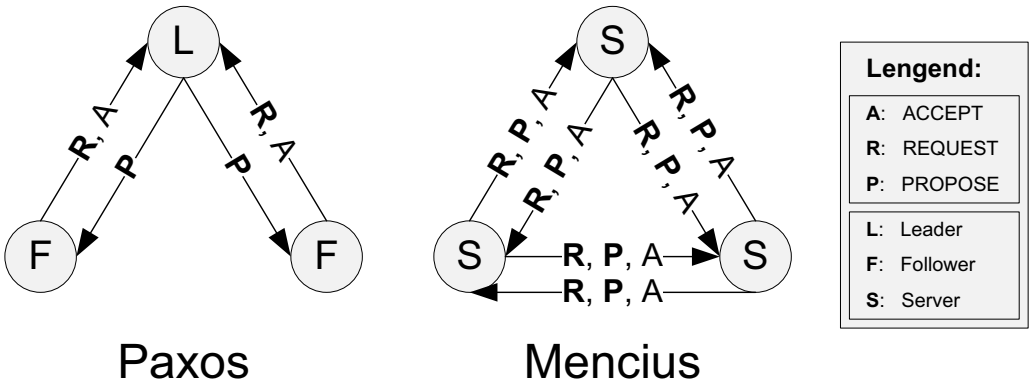


Figure 5.4: A comparison of the communication pattern in Paxos and Mencius

5.4 Implementation considerations

In the previous section, we have derived Mencius from Paxos and proved its correctness. In this section we further address issues towards a practical implementation for Mencius.

5.4.1 Prototype implementation

We implemented Mencius using C++ and used TCP as the transport protocol. The same code base was also used to implement Paxos for the performance evaluation in section 5.5. Here are some of the implementation details:

Single threaded implementation: An asynchronous and single threaded architecture was used. The architecture allows more meaningful comparison of the peak CPU-bound throughput of Paxos and Mencius on multi-core machines, since both protocol would consume a maximum of 100% CPU at peak. A multi-threaded implementation would, however, consume a maximum of 100% CPU at the bottleneck thread: the total CPU consumed by the two protocol would be different.

Nagle's algorithm: Nagle's algorithm [49] 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.

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

5.4.2 Recovery

We have already explained how Mencius recovers from broken TCP connections. The following outlines how a practical implementation of Mencius deals with process failures.

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 [15, 44], is usually application specific.

5.4.3 Revocation

So far, we have assumed that every server can start the revocation process against a suspected server. Doing so, however, is an inefficient use of resources, and it also creates a liveness problem. For example, consider three servers p, q and r. Suppose r crashes and both p and q suspect the failure. If p and q concurrently attempt to revoke r — for example, p chooses round a_1 and then q chooses round $a_2 > a_1$ before p completes — then no value will be chosen in round a_1. This situation can repeat an unbounded number of times.

This is the same liveness problem that occurs in Paxos, and it can be addressed in the same way: have a leader elected to revoke the (suspected of being faulty) server

r. Like in a state machine built with Paxos, care has to be taken in ensuring that all servers are up to date as to the point that r failed. For example, suppose r crashed after having learned the outcome of instance 2, but fails after sending LEARN to p and not to q. When p revokes r, it will start from the next instance – say, instance 5. When p prepares q to revoke r starting at instance 5, q will note that it doesn't know the outcome of instance 2. So, q can prompt p, and p can send a LEARN message that informs q as to the outcome of instance 2.

5.4.4 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 REQUEST and LEARN messages.

While a Mencius server can commit the request in just one round trip delay when there is no contention, as we have explained in section 4.4.1, extra communication delay caused by delayed commit may occur when there are concurrent suggestions. Figure 5.5 shows one such example, in which y experiences a delayed commit when it is first learned at p_1 and cannot be committed until p_1 learns x. According to Theorem 4.14, delayed commit can be up to two communication steps for Mencius, since it takes three steps for a follower in Mencius to learn the value suggested by the coordinator. 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 [34] 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.

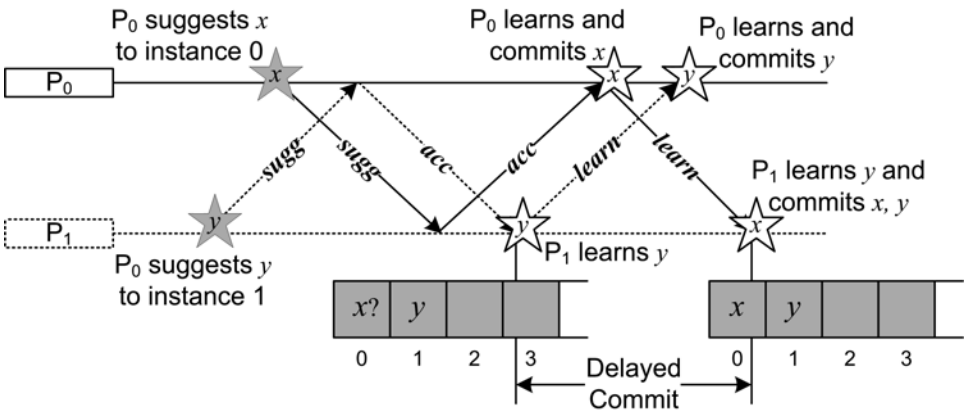


Figure 5.5: Delayed commit in Mencius.

In section 4.5.2, we discussed how to use out-of-order commit to mitigate the effects of delayed commit in protocol $\mathscr{R}$ by allowing commutable requests to be committed in any order at different servers. The same technique applies for Mencius. To support out-of-order commit, a third API IsCommute(u, v) was implemented in Mencius to allow Mencius to ask the application if two requests are commutable. This is in addition to the standard APIs Propose(v) and OnCommit(v) used for the application to issue request to Mencius and for Mencius to notify the application that a request is ready to commit.

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. We evaluate its effectiveness in Section 5.5.5.

5.4.5 Parameters

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

Accelerator M.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 M.1 generates at most 20 SKIP messages per second, given a large enough α. 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, forces the indexes of q and other servers to be 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 M.2). The communication overhead of skipping is small, as discussed in Optimization M.1 and M.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 total 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.

5.5 Evaluation

We ran controlled experiments in the DETER testbed [10] to evaluate and compare the performance of Mencius and Paxos. Our implementation of Mencius and Paxos shared the same C++ code base and used TCP as the transport protocol. We set the parameters that control Accelerator M.1 and Optimization M.3 to $\alpha = 20$ messages, $\tau = 50$ ms, and $\beta = 100,000$ instances.

5.5.1 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 5.5.4. 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 [54] 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, 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 5.5.2, 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 5.5.3. In Section 5.5.4 we show that Mencius is more scalable than Paxos. In Section 5.5.5 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.

5.5.2 Throughput

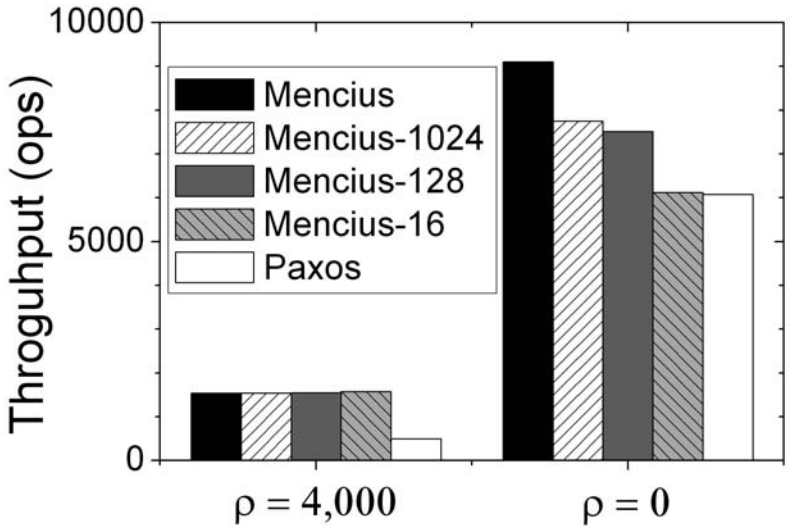


Figure 5.6: Throughput for 20 Mbps bandwidth

To measure throughput, we use a large number of clients generating requests at a high rate. Figure 5.6 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 dropped as the total number of registers decreased because with fewer registers there was more contention and dependencies to handle.

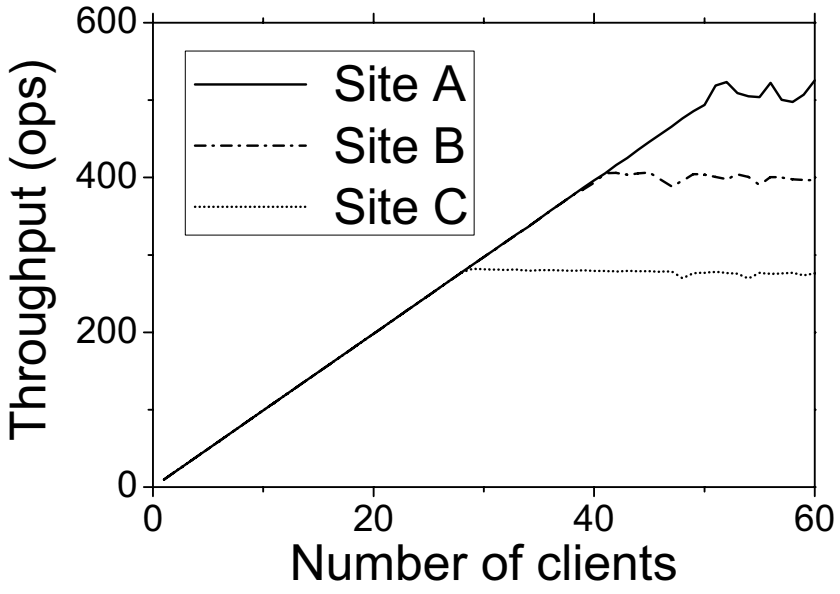


Figure 5.7: Mencius with asymmetric bandwidth ($\rho = 4,000$)

Figure 5.7 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).

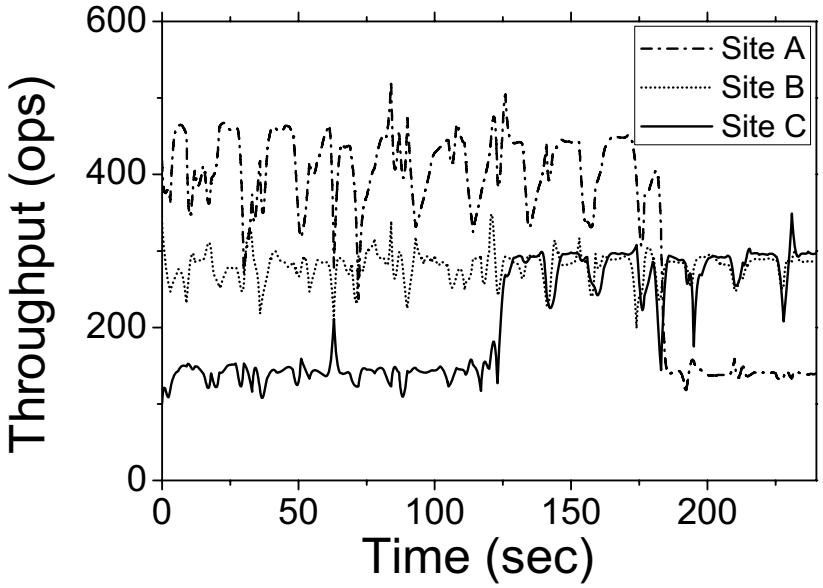


Figure 5.8: Mencius dynamically adapts to changing network bandwidth ($\rho = 4,000$)

Figure 5.8, 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.

5.5.3 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.

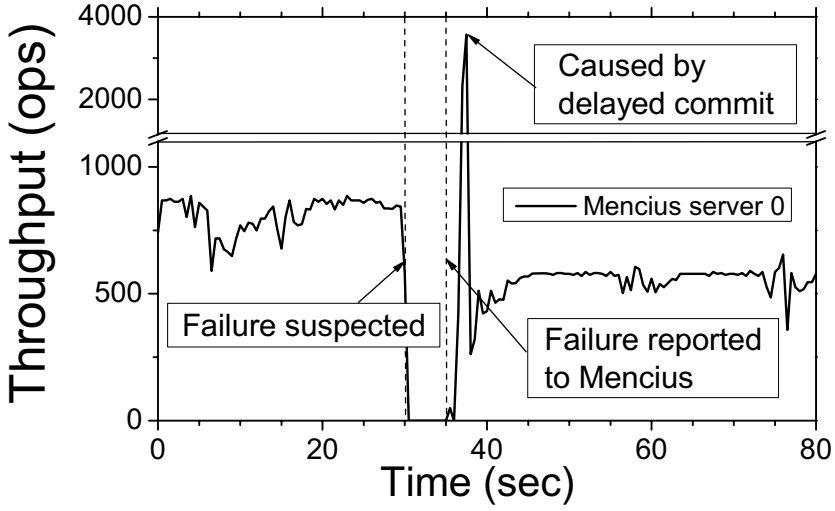


Figure 5.9: The throughput of Mencius when a server crashes

Figure 5.9 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.

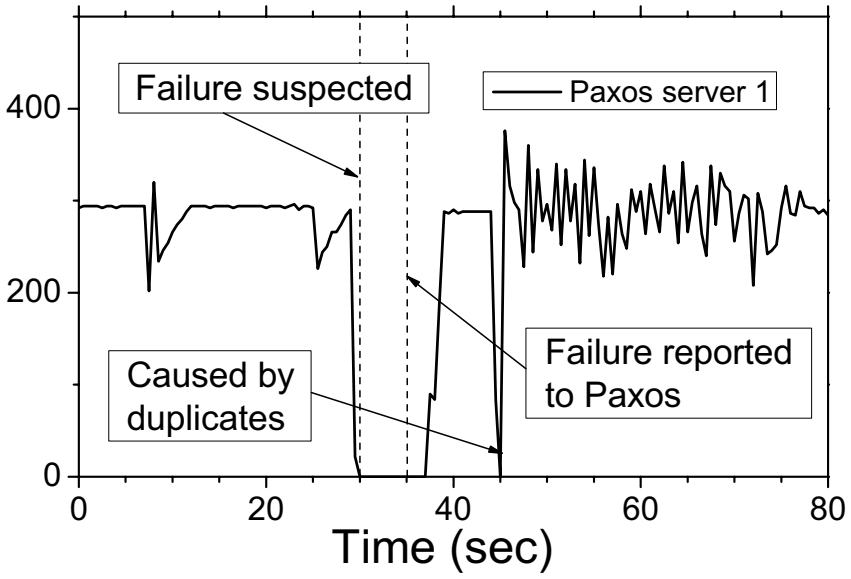


Figure 5.10: The throughput of Paxos when the leader crashes

Figure 5.10 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 ob-

served 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.

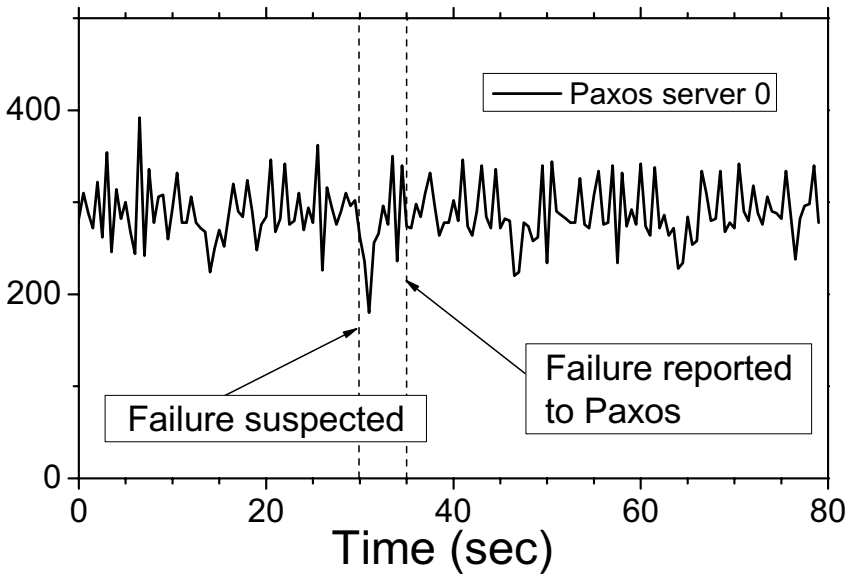


Figure 5.11: The throughput of Paxos when a follower crashes

Figure 5.11 shows Paxos's instantaneous throughput of leader p_0 when we crash p_1. There is a small transient drop in throughput but since the leader and a majority of servers remain operational, throughput quickly recovers.

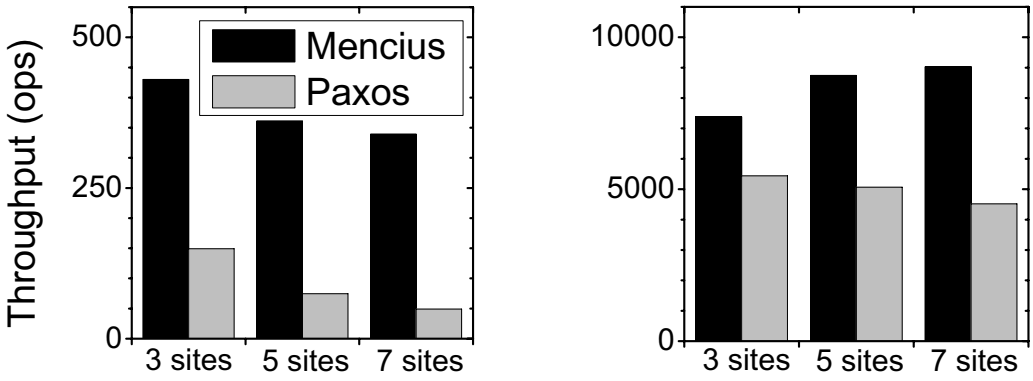
To summarize, Mencius temporarily stalls when any of the servers fails while Paxos temporarily stalls only when the leader fails. Also, the throughput of Mencius drops after a failure because of a reduction on available bandwidth, while the throughput of Paxos does not change since it does not use all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

5.5.4 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 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 5.12(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 ($79\% \approx (\frac{7}{6})/(\frac{3}{2})$)



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

Figure 5.12: The scalability of Paxos and Mencius

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 5.12(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.

5.5.5 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 exper-

iments 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.

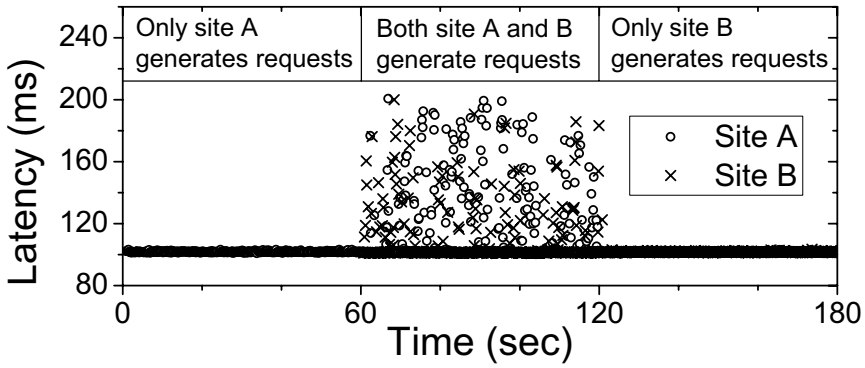


Figure 5.13: Mencius's commit latency when client load shifts from one site to another

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 5.13 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 latencies 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 5.14 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

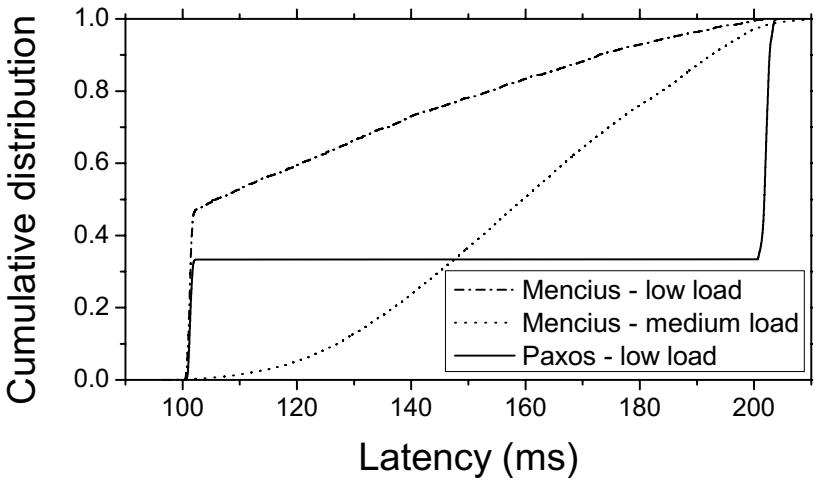


Figure 5.14: Commit latency distribution under low and medium load

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 5.15 – 5.17, 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 5.15 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.

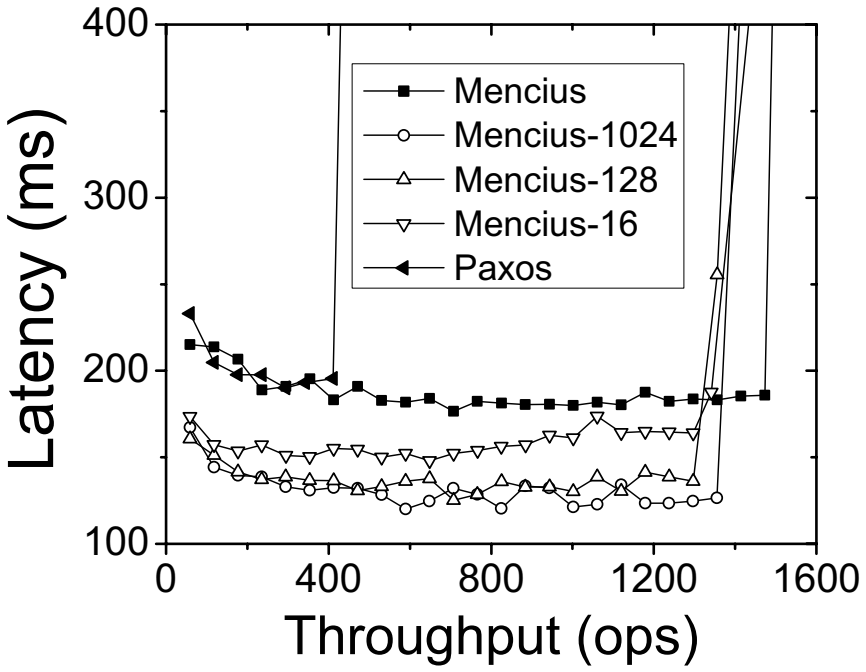


Figure 5.15: Commit latency vs offered client load ($\rho = 4,000$, no network variance)

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.

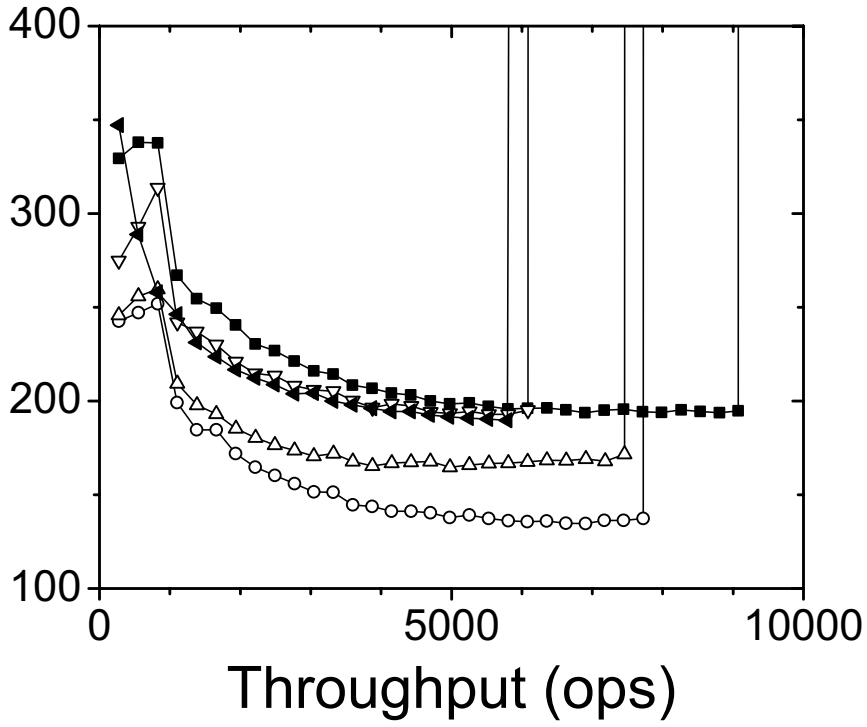


Figure 5.16: Commit latency vs offered client load ($\rho = 0$, no network varianc)

Figure 5.16 shows the result for the CPU-bound case of $\rho = 0$. It shows the same trends as Figure 5.15. 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.

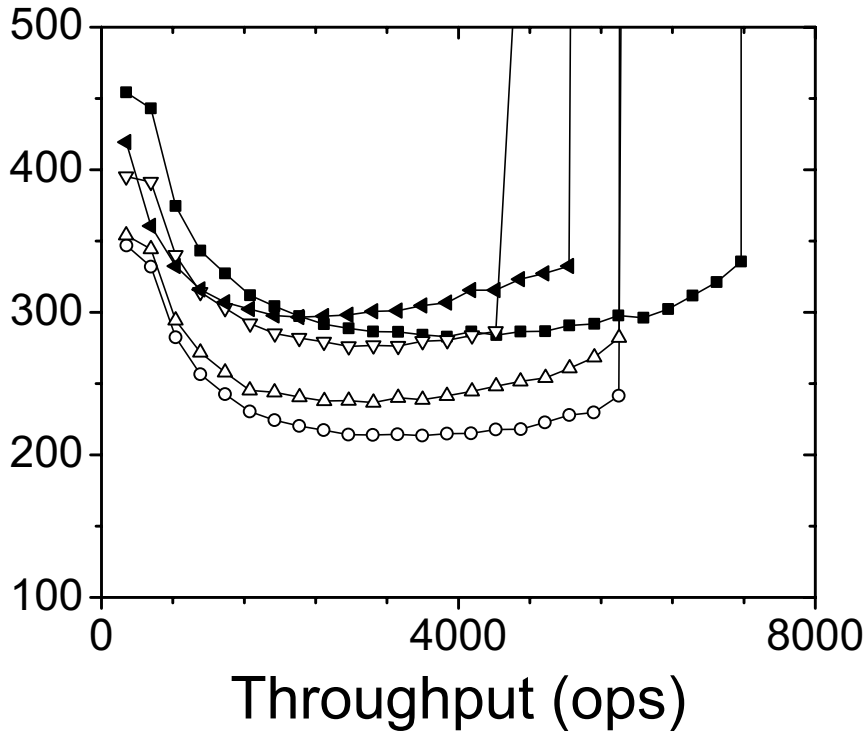


Figure 5.17: Commit latency vs offered client load ($\rho = 0$, with network variance)

In the experiment of Figure 5.17, we select delivery latencies at random. It is the same experiment as the one of Figure 5.16, except that we add a Pareto distribution to each link using the NetEm [29] utility. The average extra latency is 20 ms and the variance is 20 ms. The time correlation of the latency 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 5.17 are similar to those in Figure 5.16, despite the network variance, 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 5.16, all four Mencius variants show lower latency under lower load because delayed commit happened less often. There is no such a trend in Figure 5.17. 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 REQUEST 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.

5.5.6 Other possible optimizations

There are other ways one can increase throughput or reduce latency. One idea is to batch multiple requests into a single request, 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 requests that arrive within 50 ms into one. With small requests ($\rho = 0$), Paxos throughput increased by 4.9 and Mencius by 4.8; with large requests 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. In fact, it might reduce Paxos's throughput by increasing the wide-area message complexity.

5.6 Related work

Mencius is derived from Paxos [36, 37]. Fast Paxos, one of the variants of Paxos [39], has been designed to improve latency. However, it suffers from collisions (which results in significantly higher latency) when concurrent proposals occur. Another protocol, CoReFP [25], deals with collisions by running Paxos and Fast Paxos concurrently, but has lower throughput due to increased message complexity. Generalized Paxos [38], 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.* [24]. 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 [5] 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 [40] 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 [17].

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 [17]. For example, Zieliński presents an optimistic generic broadcast protocol that allows all messages to be delivered in two communication steps if all conflicting messages are received by different processes in the same order. Compared to Mencius, this algorithm has the disadvantages of requiring $n > 3f$ [67]. Zieliński also presents a protocol that relies on synchronized clocks to deliver messages in two communication steps [68]. 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 deterministically orders messages from multiple sources, and uses the rates at which sources produce messages to determine order, with the goal of minimizing the latency to deliver a message [3]. In contrast, Mencius does not assume knowledge of the message producing rates. It instead skip instances to reduce delivery latency. Schmidt *et al.* propose the M-Consensus problem for low latency atomic broadcast and solved it with the Collision-fast Paxos [57]. 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 different from Mencius, it assigns rounds of an M-Consensus instance to groups of proposers (collision-fast proposers), and each instance may have multiple values learned. When used in a solution to atomic broadcast, Collision-fast Paxos also suffers from a problem similar to the delayed commit problem in Mencius, since it needs to order the multiple values that are assigned to and learned in a single instance of M-Consensus. Different from Mencius, however, out-of-order commit is not possible because a slot is re-assigned to a different server upon a server failure, and different servers can propose two different arbitrary values. Recall that with Mencius, the value learned for an instance can be only the value proposed by the coordinator of the instance or *no-op*.

We are not the first to consider high-throughput consensus and fault-scalability. For example, FSR [28] is a protocol for high-throughput total-order broadcast for clusters that uses both a fixed sequencer and ring topology. PBFT [14] and Zyzzyva [33] propose practical protocols for high-throughput consensus when processes can fail arbitrarily. Q/U [2] proposes a scalable Byzantine fault-tolerant protocol.

Steward [7] 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.

5.7 Future directions and open issues

We have discussed important implementation considerations for Mencius and conducted extensive evaluation of the protocol in multi-site settings. The following issues, however, need yet to be studied in the future.

Mencius for local-area systems: Though design specifically for wide-area multi-site systems, Mencius is expected to perform well for local-area systems for the following reasons. (1) In local-area systems, the network is no longer the bottleneck, therefore, system throughput benefits from the rotating-leader design of Mencius, similarly to those CPU-bound multi-site systems. (2) Network latency has smaller variance and is more predictable in local-area system. This not only makes $\Diamond\mathbb{P}$

failure detector easier to implement but also make it less a disadvantage for Mencius to have to wait for ACCEPT messages from all correct servers before committing a request – the difference between waiting for all correct servers and only $f+1$ servers is now much smaller in local-area systems due to less variance in message latency. (3) Unless forwarding is used, Mencius in wide-area systems can only achieve the peak throughput when all sites have high load at the same time. When running in local area network, it is easy for clients to randomly select a server to send request to for achieving optimal load balancing. Despite these promising findings, we have yet to conduct experiments to further evaluate Mencius and explore possible optimization in local-area systems.

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.

Managing out-of-order commit: Out-of-order commit improves Mencius's lower latency, but also hurts its throughput. It is not clear, though, which heuristics to use to decide when to enable this feature.

Managing Nagle's algorithm: While Nagle's algorithm increases throughput when the system has high load, it also adds latency under low load. Nagle's algorithm, however, can be turned on and off on a per link basis, but it is not clear how to make such choices in an optimal way.

5.8 Summary

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

multi-site system. 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.

5.9 Acknowledgement

This chapter is, in part, reprints of material as it appears in "Mencius: Building Efficient Replicated State Machines for WANs", by Yanhua Mao, Flavio Junqueira, and Keith Marzullo, in the Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), December, 2008. The dissertation author was the primary coauthor and co-investigator of this paper.

Chapter 6

Byzantine Failure: Protocol RAM

In the previous chapter, we studied replicated state machines for multi-site wide-area systems under the crash failure model, in which the processes can only fail by stopping to execute. In this chapter, we extend the study and design a protocol called *RAM* for the Byzantine failure model, in which a process may fail in an arbitrary manner. Replicated state machine protocols that tolerate Byzantine failures are often called Byzantine fault-tolerant protocols, or BFT protocols for short.

Being able to tolerate more general failures makes BFT protocols applicable to a wider range of applications. For example, crash-failure protocols are most useful to tolerate benign hardware failures. BFT protocols, on the other hand, can be used to tolerate unexpected behavior caused by random bit flips, uncivil rational behavior caused by selfish administrators, or even attacks caused by compromised servers or clients.

Despite these advantages, BFT protocols are rarely deployed in production. Cost is one of the many reasons behind their limited use. BFT protocols require at least $3f+1$ replicas to tolerate f failures [41], f more than crash-failure protocols. Moreover, fast-learning protocols require even higher replication, for example, FaB [47] tolerates f failures with $5f+1$ replicas. Efforts that uses specialized hardware have been made to tolerate f Byzantine failures with only $2f+1$ [19,42] replicas, however, these protocols are vulnerable when the hardware is tampered with and cannot recover from tampering. We also take advantage of specialized hardware in our own protocol RAM for the purpose of reducing latency. RAM, however, does require $3f+1$ replicas so that such faulty

hardware behavior can be easily detected and any damaged rolled back and repaired.

Performance is another concern. BFT protocols usually use cryptographic primitive for message authentication and are typically more complicated than their crash-failure counterparts. As a result, they have lower throughput and higher latency even in the common case, *i.e.*, failure free execution. To counter that, many existing protocols introduce optimizations for failure-free runs. However, without careful design, such optimizations can be fragile: a single undetected faulty server can render the protocols nearly useless [20]. Designing protocols that deliver robust performance with undetected faulty servers is an active research area [6, 20]. One of the objectives of RAM to be able to deliver robust latency across wide-area multi-site systems.

Moreover, traditional BFT often apply a flat trust model to the whole system, *i.e.*, no two processes trust each other. Though simple, the flat trust model misses out practical optimization opportunities. For example, in a multi-site system, processes in the same site often have the same basic objectives since they are managed by the same administrator. Local processes are also more likely to have fate-sharing relationships since they share the same firewall and intrusion detecting system that protect the processes from outside attacks. Moreover, when processes are compromised in a site, the problem is quite severe for that site but less so for other site: masking faulty behavior only in one service is short sighted. Instead, the system administrators need to understand how widespread the attack is and restore the site's integrity. So, if desired, it is reasonable and practical for the processes in the same site to trust each other because such trust already exists in some scenarios. RAM is designed to take advantage of such trust model to optimize performance.

Finally, existing BFT protocols do not recognize different classes of failures. Indeed, BFT is essentially a masking technique: all failures up to a certain threshold are masked regardless the cause. Detecting failures, however, can improve faulty coverage and availability through repairing or excluding faulty processes. RAM is designed to expose at many failures as possible while discourage uncivil rational , selfish, and/or competitive behavior that may hurt system performance. RAM also treats different classes of failures differently. For example, a masking technique is used to cope with failures occurred at remote site; processes within a site usually trust each other, external tools is

used to detect such failures when abnormality is observed; Also, specialized hardware is usually trusted by both local and remote processes. When the hardware is compromised or tampered with, a simple mechanism is used to detect such faulty behavior. System can then be rolled back and any damage repaired after the detection of faulty behavior.

With these goal in mind, we start this chapter with section 6.1, reviewing the system model and explaining three classes of possible process behavior: correct, uncivil rational, irrational. We then state our main design goals in section 6.2. Section 6.3 discuss one of these goals: reducing latency. The results are the use of three basic techniques: **R**otating-leader design, **A**ttested append-only memory (A2M), and **M**utually Suspicous Domains (MSD) model. In section 6.4, A2M and MSD are then incorporated into a simple consensus protocol: Coordinated Byzantine Paxos, which is in turn used to build a rotating leader replicated state machine protocol in section 6.5. The resulting protocol is called *RAM*, named after the three basic techniques used. Section 6.6 and 6.7 explain how RAM copes with uncivil rational and irrational behaviors, respectively. We also implemented a prototype for RAM and report preliminary evaluation results in section 6.8. Finally, we survey related work in section 6.9 and summarize this chapter in section 6.11.

6.1 Assumptions

In this chapter, we consider a practical Byzantine failure model for the multi-site setting, meaning that processes, by default, are assumed to be able to fail arbitrarily. However, practical assumptions are also made to judiciously restricting the faulty behavior. Details about additional assumption we make can be found in section 6.3.1, 6.3.3, 6.7.1 and 6.7.2. These assumptions allows us to design efficient yet robust protocols.

Because of a well-known impassability result [41], our protocol is designed to tolerate up to $\lfloor (n-1)/3 \rfloor$ Byzantine failures. We assume minimal replication, $n = 3f + 1$ for simplicity. We also assume the availability of standard cryptographic primitives such as secure hashing [1, 52], message authentication codes (MACs) [9], and digital signatures [53]. We assume it is computational infeasible to break these primitives. Keys are assumed to have been deployed in advance, and all servers only process messages

that are properly signed or authenticated. Unless otherwise noted, we only consider faulty server behavior. Tolerating faulty client behavior is an orthogonal problem and can be dealt with using published techniques [23, 43].

We assume the existence of a *utility* for each server. For example, it could be providing low request latency for its local clients. Based on the utility, we can roughly categorize the behavior of a server into the following three broad classes:

Civil behavior: behavior that comply with the protocol specification. Civil behavior is also correct behavior.

Irrational behavior: an arbitrary behavior that can hurt the utility of a site or of the system. Irrational behavior is also Byzantine behavior.

Uncivil rational behavior: behavior in which, from an outsider's point of view appears correct. However, to seek for higher utility for its own site, the server may exploit the protocol and reorder or delay message processing. This is possible because the asynchronous message passing model does not bound the either the message delivery time or processing time. So, another server cannot tell if the extra delay is intentionally caused by the server or caused by system asynchrony. Such behavior is *uncivil* if it causes the utility of other sites to decrease. Such situations can arise from deliberate action of a site. For example, under high load conditions, an uncivil rational site can process messages from a local client in preference to those from remote clients. Other sites may not be able to determine if the additional delay is caused by uncivil behavior or network jitter.

It is easy to see that civil behavior is correct and irrational behavior is Byzantine faulty. It is, however, inherently hard to determine if a uncivil rational behavior is correct. From outsider's point of view, a uncivil rational server eventually processes and sends out message as defined by the protocol specification, also it is considered as correct by other servers. From the point view of the server in question, in which internal knowledge is available, a uncivil rational server does not comply with the protocol specification, so, it is should be considered as faulty. Figure 6.1 summarizes the relationship between correct, Byzantine faulty, civil, uncivil rational, and irrational behavior. Note

that it is possible to have rational behavior that is civil: such behavior optimizes local utility but does not hurt the utility of remote sites. Such a behavior is consider as both correct and civil.

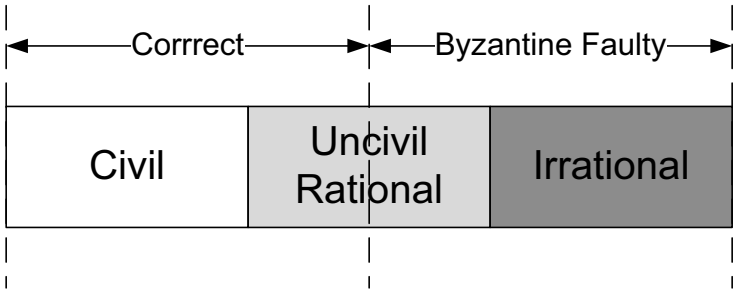


Figure 6.1: The relationship between different kinds of server behavior

Uncivil rational and irrational behavior also have different root causes. Irrational behavior is often caused by a server or a site being compromised by remote attacks that seek to disrupt the replicated state machine service. Rational behavior is often caused by faulty (selfish) local administrators that seek to increase local utility possibly at the cost of the utility of remote sites. The differences in root cause result in different strategies for coping with these two classes of behavior. The next section discusses these differences.

6.2 Design goals

In this section, we examine the three main design goals of our protocol: robust low request latency, discouraging uncivil rational behavior, and exposing irrational behavior.

6.2.1 Low latency as a main goal

In the crash failure model, we designed Mencius to have both low latency and high throughput in multi-site systems. For the weaker Byzantine failure model, we relax the performance goals to have robust low request latency and robust acceptable system

throughput.

So, we define the system *utility function* as low request latency; our main goal is to provide low request latency for multiple sites. Such low latency should also be robust. This means not only that RAM has low latency during periods of stability (no untimely communication, false suspicion, or undetected faulty servers), but also that the latency does not increase too much even in the presence of false suspicion and/or undetected faulty servers. We only require low latency when communication is timely because no protocol can be both safe and live when communication is untimely (*i.e.*, asynchronous). Reducing latency in the wide area requires protocol redesign; we discuss how to achieve this goal in section 6.3.

Throughput is a secondary objective in RAM. Indeed, the throughput of RAM is significantly lower than many existing protocols, such as [14, 33], mainly because RAM signs critical messages using digital signatures, which is in turn useful in exposing irrational behavior (see section 6.2.3). Even though throughput was not our primary goal, the throughput that RAM provides is robust and is in the range that has been sufficient for practical applications [15]. Additionally, throughput increases as hardware improves, so, we consider treating throughput as a secondary objective a reasonable engineering trade-off.

6.2.2 Discouraging uncivil rational behavior

We have discussed uncivil rational behavior, which is behavior that optimizes for local utility. Examples of such behavior including: competing for the leader role, competing for resources (by for example flooding), issuing irresponsible suspicion, omitting messages, and untimely behavior. While protocols are usually designed to achieve the best utility function for the system as a whole, such rational behavior can cause the system to diverge from that goal. It is our goal to discourage uncivil rational behavior. To achieve this goal, we designed RAM in a way such that any divergence from civil behavior should only increase the chance of reducing the local utility function. By doing so, the best strategy for any uncivil rational server is to follow the protocol.

6.2.3 Exposing irrational behavior

Most existing BFT protocols focus, naturally, on masking Byzantine failures. By doing so, an undetected faulty server can cause significant reduction to system performance [6, 20]. For example, a faulty leader in PBFT can increase the latency for all servers significantly by intentionally delaying requests just enough not to trigger timeouts. While not all failures can be pinpointed, detecting those that can be is useful. So, we designed RAM to expose as much irrational behavior as possible. For example, RAM uses digital signature to sign critical messages. Though resulting lower throughput when comparing to the much cheaper MACs, digital signature does provide the ability for a server to prove the faulty behavior of another server to a third party, which is impossible if MACs are used instead.

In addition, RAM also makes practical assumptions that (1) a client trusts its local server and (2) servers trust each other's A2M. Should these assumptions be broken — a sign of site compromise — RAM will temporarily enter an unsafe state, but will detect the failure quickly. Once detected, more broad site recovery tools can be used to roll back and repair any damage.

6.3 Reducing latency for PBFT

We have explained Paxos and PBFT in section 2.7 and 2.8 respectively. Since reducing latency is our primary goal, we uses a variant of Paxos that broadcasts the ACCEPT message for reducing latency (see Figure 2.5). 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 when 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-PREPARE and 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. This is because a PBFT client does not trust its

local server. Waiting for $f+1$ matching requests ensures at least one of them is from a correct server. In the remainder of this section, we explore techniques to reduce latency for PBFT in multi-site systems.

6.3.1 Mutually Suspicious Domains

PBFT assumes a flat Byzantine model, in which there is no trust between pairs of processes: both clients and correct servers have to be aware of Byzantine servers. Applying the flat 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 and the servers have incentives to improve the utility for its local clients. (2) Client and server machines in the same site are more likely to share vulnerabilities because 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 to recognize the fact that a local server is more trustworthy than remote ones.

Definition 6.1 (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 [30]. 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.

6.3.2 Rotating leader design

PBFT, as well as other more recent BFT protocols [20, 33], 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 remote sites. With MSD, clients trust their local servers. Consequently, it is natural for a server to propose requests on behalf of its local clients. This way, the clients do not have to worry about the server being unfair. Naturally, this leads to the adoption of the rotating leader design scheme as described by the generic protocol $\mathscr{R}$.

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. Concurrent requests may cause the delayed commit problem and increase the latency by up to one step. This extra latency can be reduced by allowing out-of-order commit to safely commit commutable requests at different servers in different orders. We evaluate effectiveness of out-of-order commit for RAM in section 6.8.3.

Finally, being the leader in RAM is a privilege rather than a right: the leader status of a server may be revoked if it is acting suspiciously. We designed RAM such that divergence from the correct behavior only increases the probability of a server being revoked, which in turn leads to lower local utility. As a result, uncivil rational behavior is discouraged because the best strategy for a rational server is to follow the correct behavior.

6.3.3 A2M for identical Byzantine failure

While it only takes Paxos one step (PROPOSE) for the servers to confirm the value proposed by the leader, it takes PBFT two steps (PRE-PREPARE and PREPARE) to confirm the leader's proposal. The extra step 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 sub-protocol using the extra PREPARE phase, PBFT is able to simulate identical Byzantine failure [8]: if a server has prepared a value from the leader, all other servers will prepare either the same value or no value.

Definition 6.2 (Attested append-only memory (A2M)): A tamper-resistent hardware that function as an append-only log: when asked, A2M appends and digitally signs a new log entry if and only if it is consistent with previous entries in the log [19].

Conceptually, A2M contains an internal logic that determines consistency and an unbounded number of log entries. Since any practical implementation of A2M can only support a bounded number of log entries, an application that uses A2M is also allowed to discard the entries that are no longer useful. The logic that determines consistency is usually application specific and can be quite complicated. TrInc [42], a simplified version of A2M, can be used to offload much of the logic to the application and only stores a $\langle counter, value \rangle$ pair in the log.

A2M has been proposed to design BFT protocols that tolerates f failures with only $2f+1$ replicas. This is done by pairing each server with an A2M which is treated as a trusted third party by all servers. Each server is required to record all outgoing messages into its A2M, which in turn needs to make sure they are consistent with respect to the semantics of the BFT protocol. Such a usage of A2M reduces the degree of replication, but it has its drawbacks: non-faulty replicas must both log/attest all protocol messages before sending them and can only handle one request at a time. The logic of A2M is also complicated, and so prone to error and vulnerability. Relying A2M to tolerate f failures with $2f+1$ replicas also makes the system vulnerable to a compromised A2M. Though highly unlikely, when f A2Ms are compromised, it is impossible to determine the faulty party or to recovery from the damage caused by such compromised A2Ms.

We instead opted for a higher degree of replication and simpler design. Requiring $3f+1$ replicas allows us to detect faulty A2Ms and recover from the damage caused by them. This is useful in environments that cannot fully guarantee the integrity of A2M devices. We discuss this topic further in section 6.7.1.

Additionally, we only uses A2M to implement identical Byzantine failure with only one wide-area message delay. Each server is paired with a local A2M. 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 records the last pair of sequence number and value it has signed and only signs a new proposal if it is immediately succeed the record sequence: this way the server cannot have the A2M signing inconsistent proposals or leaving gaps in the consensus sequence space.

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: all consensus based replicated state machine protocol needs to propose a request at one point or another.

6.4 Coordinated Byzantine Paxos

RAM is a replicated state machine protocol that executes an unbounded sequence of simple consensus instances. Each instance of simple consensus is solved using Coordinated Byzantine Paxos, which is a Paxos-like protocol that incorporates MSD and A2M to tolerate f Byzantine failures using $3f+1$ replicas.

Like Paxos, Coordinated Byzantine Paxos operates in rounds. Each replica is responsible for a unbounded number of rounds and acts as the leader of these rounds. This implies that no two replicas are responsible for the same round. The goal of the replicas is to have a value chosen for a round in which it is the leader. A value v is chosen in a round r if and only if $2f+1$ replicas accepted v in round r. Coordinated Byzantine Paxos ensure that once a value v is chosen in round r, the replica responsible for round r' ($r' > r$) can and will only proposes v for round r'. This ensures that the

Table 6.1: Summary of messages in Coordinated Byzantine Paxos.

Message	Meaning
REQUEST	A Client sends a request to a replica
SUGGEST	The coordinator proposes a client request
ACCEPT	A Replica accepts the proposal
SKIP	The coordinator proposes *no-op*
SU-RELAY	A replica relays the SUGGESTmessages
REPLY	A replica sends the result of a request
SK-RELAY	A replica relays the SKIP messages
O-REPLY	Optional reply certificate
NEW-LEADER	The new leader initiate the leader change
ACK	A replica acknowledges to NEW-LEADER
PRE-PREPARE	The new leader proposes a value based on the progress certificate
PREPARE	A replica confirms the value proposed by the new leader

protocol is safe.

Coordinated Byzantine Paxos incorporates MSD and A2M. This means that the clients always trust their local server and all the servers trust the A2M. Critical messages such as SUGGEST and SKIP (and by extension SU-RELAY and SK-RELAY) must be signed by one A2M. Like Coordinated Paxos, all servers with Coordinated Byzantine Paxos start with an initial round r_0, in which the coordinator is the leader. Also similar to Coordinated Paxos, a server with Coordinated Byzantine Paxos can invoke one of the three actions given below, based on its role. The pseudo code of Coordinated Byzantine Paxos can be found in Appendix D. We uses the following notation: (1)

- $\langle m \rangle_{\alpha_p}$: a message m from p authenticated using the shared secret keys between p and other processes.
- $\langle m \rangle_{\sigma_p}$: a message m signed by server p.
- $\langle m \rangle_{\rho_p}$: a message m signed by the A2M of server p.

Suggest: The coordinator c can *suggest* by proposing a client request to the initial round r_0. It does this by broadcasting a $\langle \text{SUGGEST}, v \rangle_{\rho_c}$ message, which is signed by its local A2M. Upon receiving the SUGGEST message for the first time, a

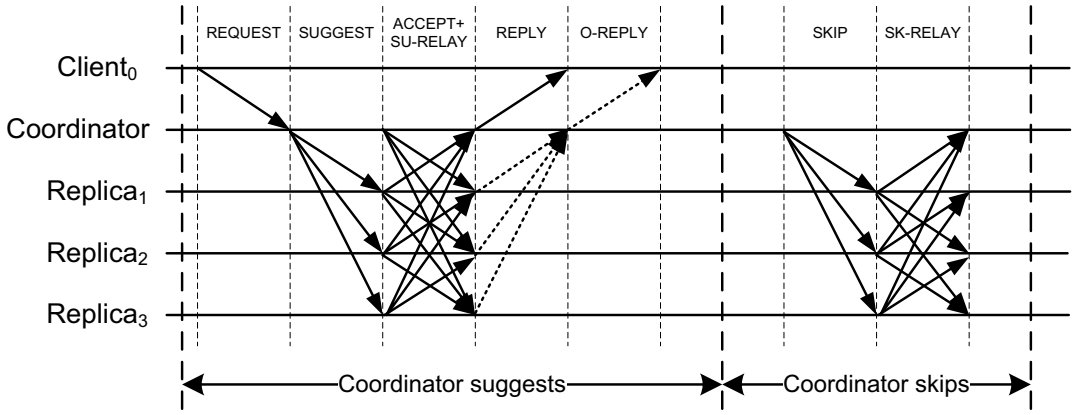


Figure 6.2: A space time diagram showing the message flows of the suggest and skip actions in Coordinated Byzantine Paxos.

server p accepts it unless it previously promised not to do so. It then broadcasts a $\langle \text{ACCEPT}, v, r_0 \rangle_{\alpha_p}$ message and also relays the SUGGEST message using SU-RELAY[1] as a separate message. A server p learns the outcome when it receives $2f+1$ matching ACCEPT messages. The coordinator c commits and executes the request to generate the reply R. Then it sends a $\langle \text{REPLY}, R \rangle_{\alpha_c}$ message to the client. The client learns the outcome of the its request once it receives the REPLY from the coordinator, which is its local server. Instance 0 in Figure 6.2 summarizes the message flow of suggest. The dotted lines in this figure represent optional REPLY and O-REPLY messages, which are useful for detecting a faulty local server. We further explain them in section 6.7.2.

Skip: The coordinator c can *skip* by proposing *no-op*. It does this by broadcasting $\langle \text{SKIP} \rangle_{\rho_c}$, which is signed by its local A2M. Note that no ACCEPT message is generated to acknowledge the SKIP message because Coordinated Byzantine Paxos is skipping fast. Upon receiving the SKIP message for the first time, a server relay the message using SK-RELAY to all other servers and learns *no-op* has been chosen. A server can learn *no-op* immediately because *no-op* is the only possible

[1] A SU-RELAY message has exact same fields and content as the SUGGEST message being relayed. We give it a new name here to distinguish it from the original SUGGEST message. For the same reason, we later use SK-RELAY to distinguish a relayed SKIP message from the original SKIP

outcome: the coordinator has proposed *no-op* and all the other servers can only propose *no-op* by the definition of simple consensus. Instance 1 in Figure 6.2 summarizes the message flow of skip.

Revoke: For some protocol parameter α ($1 \leq \alpha \leq f+1$), if $f+\alpha$ servers suspect the current leader is faulty, then a new leader is elected to revoke the suspected leader. Figure 6.3 summarizes the message flow of revoke. We further explain the details for the revocation sub-protocol in section 6.4.1. The new leader does this by asking all servers to advance to round a higher round r' and polling all the other servers using NEW-LEADER message. Upon receiving this the replicas advance to round r', promise not to accept any value for round $r^* < r'$, and response with a signed ACK message. The ACK message also includes the value a server has accepted and the round in which the value was accepted. The new leader then gathers $2f+1$ ACK messages to form a *progress certificate*. If the progress certificate indicates that some value v might have been chosen, then the new leader proposes v; otherwise it proposes *no-op*. The subsequent execution is similar to that of PBFT.

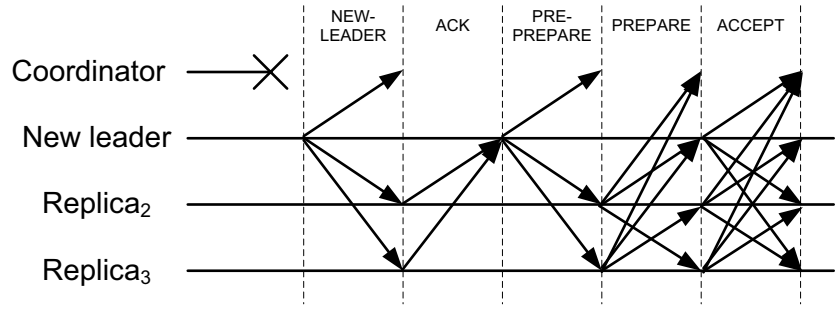


Figure 6.3: A space time diagram showing the message flow of the revoke action in Coordinated Byzantine Paxos.

Note that $\alpha+f$ suspicion is required to revoke a server. $\alpha \geq 1$ ensures at least one correct server suspects the current leader, and $\alpha \leq f+1$ because up to f servers may be faulty: the protocol can not wait for more than $2f+1$ suspicions. Choosing a

large α reduces the probability of unnecessary revocation caused by network jitter. We discuss this in section 6.6.1.

Table 6.1 summaries the messages used in Coordinated Byzantine Paxos. SUGGEST and SKIP message are initialed by the coordinator and are in the critical execution path. So, they are signed by A2M to enable identical Byzantine failure. they are also relayed with SU-RELAY and SK-RELAY respectively to cope with faulty servers that omit the SUGGEST and SKIP message to some of the servers. Relaying ensures that if one correct servers receive the messages all correct servers will eventually do. This helps to bound the latency increase that irrational servers can cause (see section 6.7.4).

The ACK messages are digitally signed by the replying servers (and not by the server's A2M). A digital signature is required so that a new leader can later prove that the value it proposes is indeed a valid choice. An alternative is to have the servers broadcast ACK messages authenticated by MACs. The servers then only accept the new-leader's proposal if they have received sufficient number of ACK messages that can be used to construct a progress certificate and validate the new proposal. Since revocation is usually not in the critical execution path of Coordinated Byzantine Paxos and RAM, we opt for the simpler, although more expensive, alternative of using digital signature.

6.4.1 Revocation sub-protocol

We assume an external leader election protocol that is at least as strong as Ω is used for the revocation sub-protocol. When the leader l of the current round r is suspected of having failed, a server p sends a $\langle \text{SUSPECT}, l, r, L \rangle_{\sigma_p}$ to a newly elected leader L. L gathers $f+\alpha$ such SUSPECT messages to construct a revocation certificate rc. L then broadcasts $\langle \text{NEW-LEADER}, rc, r' \rangle_{\alpha_L}$, for some round $r' > r$. If a server p receives a valid NEW-LEADER message and is currently in a round r_p that is smaller than r', p advances to round r' and acknowledges with an $\langle \text{ACK}, r', v_p, \sigma_{v_p}, r_p \rangle_{\sigma_p}$ message. v_p is the value that p is currently accepting in round r_p. σ_{v_p} is the necessary signature for other servers to validate v_p. When $r_p = r_0$, σ_{v_p} is the one signed by the A2M of the coordinator. Otherwise, it is a collection of $2f+1$ signatures signed in the prepare phase of round r_p, as described below.

The leader L then collects $2f+1$ ACK messages to construct a progress certifi-

cate pc, based on which, it picks a value v' to propose for round r' using the following rules:

1. If the ACK messages indicate some server has accepted some value, then L picks the value v with the highest round number in these ACK messages.

2. If the ACK message indicate no server has accepted any value, then L picks *no-op*.

Once a value v' is picked, L broadcasts a $\langle \text{PRE-PREPARE}, pc, v', r' \rangle_{\sigma_L}$. A server p only processes a PRE-PREPARE message if (1) p is currently in round no larger than r'; (2) v' is the valid choice according to pc; and (3) p has not received any other valid PRE-PREPARE messages for round r'. p acknowledges a valid PRE-PREPARE message by broadcasting a $\langle \text{PREPARE}, pc, v', r' \rangle_{\sigma_p}$ message. A server p then collects $2f+1$ matching valid PREPARE message for round r' before accepting v'. It does so by broadcasting an $\langle \text{ACCEPT}, v', r' \rangle_{\alpha_p}$. Note that, if no value is chosen in this round and p later needs to advance to a higher round r'' upon receiving a NEW-LEADER message from the leader in round r'', p responses with an $\langle \text{ACK}, r'', v', \sigma_{v'}, r' \rangle_{\sigma_p}$ message as we described earlier. $\sigma_{v'}$, the signature that vouches for v', is the signatures of the $2f+1$ matching PREPARE messages p received in round r'. Upon receiving the ACK message, the new leader in round r'' can reconstruct these $2f+1$ PREPARE messages and validate the signatures.

After sending out the ACCEPT messages, server p then collects matching ACCEPT messages. Once $2f+1$ matching ACCEPT messages are collected for round r', p learns v' and commit it. A reply is then sent to the client if necessary.

6.4.2 Correctness proof

Lemma 6.1: Assuming A2Ms can be trusted and the MSD model, Coordinated Byzantine Paxos satisfies Validity (SC.2), Integrity (SC.3), and Non-triviality (SC.5).

Proof. Assuming MSD allows the clients to trust their local server, and so it is safe for the client to commit to a reply upon receiving it from its local server.

Validity (SC.2): The only case in which validity applies is when the coordinator skips, *i.e.*, proposes *no-op*. In this case *no-op* is the only possible consensus outcome because: (1) All correct servers learns *no-op* upon receiving the skip message

from the coordinator; and (2) Any subsequent revocation can only proposes *no-op*. This is because the coordinator's A2M ensures the coordinator cannot submit any value other than *no-op*. And so any subsequent progress certificate can only vouch for *no-op* or no value. In either case, *no-op* will be proposed by the new leader.

Integrity (SC.3): This is trivially satisfied.

Non-triviality (SC.5): This can be verified: (1) When the coordinator proposes a client request v, v will be the outcome if the coordinator is correct and no server suspects the coordinator; and (2) We have explained *no-op* is the only possible outcome if the coordinator proposes *no-op*.

□

Lemma 6.2: Assuming A2Ms can be trusted, no two correct servers p and q learn different values in the same round r.

Proof. We prove the lemma by verifying the following two cases cases:

- $r = r_0$: In round r_0 a server can only accept the proposal from the coordinator. Since the A2M prevents the coordinator from proposing two values, no server can accept or learn different values.

- $r \neq r_0$: p and q must each receive ACCEPT messages for round r from a quorum of $2f+1$ servers. Since we have $3f+1$ server in total, the two quorums have at least $f+1$ servers in common. At least one of these $f+1$ servers is a correct one, since we have at most f faulty servers. It is impossible for p and q to accept different values in round r because a correct server does not send conflicting ACCEPT messages.

□

Lemma 6.3: Assuming A2Ms can be trusted, Coordinated Byzantine Paxos satisfies Agreement (SC.4).

Proof. From Lemma 6.2, we have already ruled out the possibility for two servers to learn different values in the same round. So, without losing generality, we assume two server p and q learn distinct value u and v in round r_1 and r_2 respectively. We also assume r_1 is the smallest round any server learns u; r_2 is the smallest round any server learns v; and $r_1 < r_2$.

If $r_1 = r_0$ and $u = \textit{no-op}$, we already explained that *no-op* is the only possible consensus outcome, so, the Agreement property is satisfied.

Otherwise, $r_1 \neq r_0 \vee u \neq \textit{no-op}$, p must have received $2f+1$ ACCEPT messages in round r_1 for value u. At least $f+1$ such ACCEPT messages are from correct servers. We use S to denote these $f+1$ correct servers.

Obviously $r_2 > r_1 \geq r_0$, q must have learned v through revocation: a server learns a value in a round larger than r_0 only through revocation. This means q has received matching ACCEPT messages from $2f+1$ servers. We use T to denote these $f+1$ servers. Since $|S| = f+1$ and $|T| = 2f+1$, S and T intersect on at least one correct server. So, at least one correct server accepted v in round r_2. This correct server must have received matching PREPARE messages for v in round r_2 from $2f+1$ servers, which we denote as T'. S and T' also intersect on at least one correct server. So, there exists at least one correct server that accepted u in round r_1 and later prepared v in round r_2.

Now, let $r' > r$, be the smallest round in which a correct server p' does the preceding, *i.e.*, p' accepted u in round r_1 and prepared v in round r'. For p' to have done this, the leader of round r' must have constructed a progress certificate that vouched for v. The progress certificate consists of ACK messages from a set of $2f+1$ servers, say T^*. Since the certificate vouches for v, the ACK message with the highest round number in the certificate must have the form $\langle \text{ACK}, r', v, \sigma_v, r^* \rangle_{\sigma_s}$, *i.e.*, s, the sender of the ACK message, claims that it has accepted v in some round r^*.

We argue this is impossible because:

- If $r^* < r_1$, then all correct servers in T^* sent ACK messages indicates that they have not accepted any value in rounds between r^* and r'. S and T^* intersects on at least one correct server. This correct server must have not accepted any value in round r_1, since $r^* < r_1 < r'$ and the correct server advances to round r' before sending

back the ACK message. This contradicts with the assumption that all servers in S accepted u in round r_1.

- If $r^* = r_1$, there are two cases:

  1. $r_1 = r_0$: This means the coordinator has proposed u in round r_0. It is impossible for s to collect σ_v because the trusted A2M would not have signed any value other than u.

  2. $r_1 > r_0$: Since u is accepted by $f+1$ correct servers in round r_1, they must have received $2f+1$ matching PREPARE messages for value u. At least $f+1$ PREPARE messages are from correct servers. This made it impossible for s to collect $2f+1$ matching signatures to construct σ_v for v in round r_1: it can collect at most $2f$ signatures since the rest $f+1$ servers has already prepared for u.

- If $r^* > r_1$, then $2f+1$ servers must have sent the PREPARE messages for value v in round r^*. At least $f+1$ of these servers are correct, which contradicts with the assumption that $r' > r*$ is the smallest round for a correct server to do so.

So, we conclude that Coordinated Byzantine Paxos satisfies the agreement property.

□

Lemma 6.4: Assuming faulty coordinator is eventually suspected and a failure detector that is at least as strong as Ω is used to select the new leader for revocation, Coordinated Byzantine Paxos satisfies Termination SC.1.

Proof. If the revocation process is not involved, then the coordinator must be correct. In this case, Coordinated Byzantine Paxos terminates eventually once the coordinator sends out its proposal to round r_0. This is true because every step of the protocol only need the responses from $2f+1$ replicas, which a correct server eventually receive because at most f servers are faulty and the SUGGEST and SKIP messages are relayed.

If the revocation process is involved, eventually exactly one server will lead the revocation process in round r. The leader can eventually collect the progress certificate for round r because it eventually receives the ACK messages from the $2f+1$

correct servers. After that all correct server will eventually receive $2f+1$ matching PRE-PREPARE and PREPARE messages in round r. This will generate $2f+1$ matching ACCEPT message for round r because no other server will be elected to start a higher round since we assume Ω is used to elected the leader. Eventually, every correct server receives these $2f+1$ matching ACCEPT messages, learns the simple consensus outcome, and terminates consensus. □

Theorem 6.5: Assuming (1) A2Ms can be trusted; (2) the MSD model; (3) faulty coordinator is eventually suspected; and (4) a failure detector that is at least as strong as Ω is used to select the new leader for revocation, Coordinated Byzantine Paxos implements simple consensus.

Proof. From Lemma 6.1, 6.3,and 6.4. □

6.4.3 Performance metrics

It is not difficult to see that the quorum size for Coordinated Byzantine Paxos is $2f+1$. The message complexity of suggest is $2n^2-4n+4$: one REQUEST message from the client to the coordinator, $n-1$ SUGGEST messages, $(n-1)(n-2)$ SU-RELAY messages, and $(n-1)n$ ACCEPT messages. This total does not count either the $n-1$ optional REPLY messages from the followers to the coordinator or the O-REPLY message from the coordinator to the client. The message complexity of the skip action is $(n-1)^2$: $n-1$ skip messages and $(n-1)(n-2)$ SK-RELAY messages. The message complexity of revocation is higher than that of suggest, and depends on whether concurrent leaders were elected for revocation. Since revocation cost is amortized in the long run (see section 6.5.3), we do not analyze the message complexity of revocation further.

During failure-free runs, it takes two wide-area communication steps to learn a suggestion and one step to learn a skip. If the coordinator is faulty and omits SUGGEST or SKIP messages to some of the servers, then up to one additional step may be needed for all the servers to learn the outcome: relaying guarantees that all correct server receives the SUGGEST or SKIP messages two steps after the coordinator sends them to one correct server. Revocation takes at least five steps: this is the case when no concurrent leaders are elected for the revocation process.

6.5 Deriving RAM

We have explained Coordinated Byzantine Paxos, a simple consensus protocol that assumes A2M and the MSD model. We now apply Coordinated Byzantine Paxos to the generic rotating leader protocol $\mathscr{R}$ to obtain RAM. RAM uses the rotating leader scheme defined in Definition 4.2 and use Coordinated Byzantine Paxos to solve each instance of simple consensus. The execution of RAM are defined by a set of rules and optimizations. As we did with Mencius, we introduce the set of rules for building a simple yet not efficient called protocol $\mathscr{Q}$. We then discuss the set of optimizations for obtaining a more efficient protocol. We call this final protocol RAM. Many of the rules and optimizations are similar to that of $\mathscr{R}$. We restate these rules and optimizations here for the purpose of completeness and point out the differences.

6.5.1 Revocation in RAM

Revocation is part of core mechanisms of the rotating-leader design. Before discussing the detail rules for RAM, we explain some of the design decision we made for the revocation process.

With the rotating leader scheme, faulty servers are revoked to allow correct servers to make progress, *i.e.*, to ensure liveness. Such a process involves the use of failure detector. We already know that the weakest failure detector for implementing consensus is Ω, which provide some sort of eventually accuracy. This is also true for $\Diamond\mathbb{W}$, a class of failure detector that has been shown to be equivalent to Ω: $\Diamond\mathbb{W}$ provide eventually weak accuracy. While it is not difficult to implement such classes of failure detectors, 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. Any rotating leader protocol, *e.g.*, Mencius and RAM, needs to properly handle false suspicion to be efficient.

When false suspicion happens with Mencius, a server can resume action by setting its index to be greater than the last instance that was revoked. A falsely suspected server makes other servers update their indexes and skip their unused turns (see Rule M.2). This quickly synchronizes the indexes of the servers and allow all correct

servers to make progress. Mencius has no time period associated with revocation because a revoked servers has either crashed, or has been falsely suspected and so should leave revocation immediately.

The strategy of Mencius, however, does not work with RAM which considers Byzantine failures: a Byzantine faulty server can abuse it to nullify the effects of revocation. This is reflected in the additional conditions of Rule R.1 – R.4 (see next section) that prevent a suspected server from immediately getting out of being revoked, regardless if the suspicion is a false positive.

Because RAM, by design, does not allow falsely suspected server to end revocation immediately, revocation in RAM has time period associated with it, *i.e.*, it is set to expire unless otherwise renewed. This allows falsely suspected server to eventually resume action. We further discuss this topic in section 6.6.1.

Even though it cannot suggest requests on behalf of its clients, a falsely suspected server can either wait out the revocation period or forward its requests to other servers expecting that a correct server will propose them on its behalf.

6.5.2 Protocol $\mathscr{2}$

Rule R.1: Each server p maintains its next available simple consensus instance number in a variable I_p, which we call p's index. Upon receiving a new client request r, if server p is not currently under revocation, p proposes r to I_p and updates I_p by assigning it the sequence number of the next instance it coordinates.

Rule R.2: When a server p receives a SUGGEST message from server q for an instance i greater than I_p, if q is not currently being revoked, *i.e.*, if p does not know q is being revoked, p skips its turns for instances between I_p and i by proposing *no-op*s and updates I_p accordingly; otherwise p ignores the SUGGEST message.

Rule R.3: Let q be a server that $f + \alpha$ servers suspects has failed. A non-faulty server p is elected to revoke q. 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.

Rule R.4: When a server p suggests a value $v \neq$ *no-op* to instance i but it learns *no-op* for instance i, then p suggests v again when p is not currently being revoked.

It is straightforward to see that Rule R.1 – R.4 are similar to Rule RL.1 – RL.4 of $\mathscr{R}$. Indeed, Rule R.1, and R.4 are the same as Rule RL.1 and RL.4 except that the former exclude server p from action when p is currently being revoked. Rule R.2 is the same as Rule RL.2, except Rule R.2 requires that a server p only skips its turns when the SUGGEST message is from a server that is not currently being revoked; Rule R.3 is the same as Rule RL.3, except Rule R.3 requires $f + \alpha$ suspicion for revocation.

We have explained earlier the additional condition of Rule R.3. The additional conditions of Rule R.1 and R.4 are in place to prevent a falsely suspected correct server from ignoring a revocation against itself. This is in place because our protocol, by design, punishes servers that ignore revocation. Following R.1 and R.4 helps to prevent unnecessary penalty for falsely suspected servers. We discuss this further in section 6.7.3. The additional conditions of Rule R.2 is in place to prevent a revoked server from easily getting out of revocation.

Lemma 6.6: $\mathscr{Q}$ with Rule R.1 – R.4 is always safe.

Proof. From Lemma 4.6 we know that Rule R.1 – R.4 without the aforementioned additional conditions are safe. The additional conditions merely restricts the actions that $\mathscr{Q}$ can take, in other words, the possible behavior of $\mathscr{Q}$ is more restricted and is a subset of the possible behavior without such conditions. So, $\mathscr{Q}$ is safe with such conditions. □

Lemma 6.7: Assuming $\Diamond\mathbb{P}$, $\mathscr{Q}$ with Rule R.1 – R.4 is live.

Proof. From Lemma 4.8 we know that Rule R.1 – R.4 without the aforementioned additional conditions are live. $\Diamond\mathbb{P}$ guarantees that eventually all the and only the crashed servers are suspected. We prove the liveness of $\mathscr{Q}$ by argue that the additional conditions eventually do not apply when assuming $\Diamond\mathbb{P}$: the eventual execution of the actions in Rule R.1 – R.4 guarantees the liveness of the protocol.

Rule R.1 and R.4: The additional conditions exclude a revoked server p from suggesting or re-suggesting requests. If p is faulty, then it does not matter, since only requests sent to the correct server are required to be eventually committed. If p is correct, then false suspicions of p eventually cease, after which time, p can take on the action of Rule R.1 and R.4 if necessary.

Rule R.2: The additional condition have server p ignore the SUGGEST messages from a revoked server q. If q is faulty, then it does not matter, since only requests sent to the correct server are required to be eventually committed. If q is correct, then false suspicions of q eventually cease, at which time, q can take on the action of R.4 to re-suggest the request if necessary.

Rule R.3: The additional condition make it less likely to trigger the revocation because of the additional suspicion required. However, since we assume $\Diamond\mathbb{P}$, any faulty server will be eventually suspected by all correct servers, the number of which is at least $2f+1$. Since $\alpha \leq f+1$, a faulty server is eventually suspected by sufficient number of servers, and hence revoked. Since the protocol only relies on the eventually revocation of faulty servers to provide liveness, the additional conditions of Rule R.3 does not affect liveness.

□

Theorem 6.8: Assuming $\Diamond\mathbb{P}$, $\mathscr{Q}$ with Rule R.1 – R.4 implements replicated state machines.

Proof. From Lemma 6.6 and Lemma 6.7. □

We argue that it is not difficult to implement $\Diamond\mathbb{P}$ for $\mathscr{Q}$. $\Diamond\mathbb{P}$ is only required for liveness, which in turn only requires requests submitted by the correct servers to be eventually committed. With $\mathscr{Q}$, each correct server is responsible for suggesting its own requests. A request will eventually be learned in whichever simple consensus instance it is submitted to, as long as the server is not falsely suspected. This condition eventually holds because we assume $\Diamond\mathbb{P}$. So, a faulty server can only prevent learned request from committing by creating gaps in the consensus sequence. This can be done in two ways: by not skipping turns or by not suggesting requests. A faulty server can only do this by not sending the SKIP or SUGGEST message to none of the correct servers because the relay mechanism ensures if one correct server receives such messages, all correct server will. This makes it easy to detect such faulty behavior using timeouts. Eventually accuracy can be obtained by exponentially increasing timeout. To implement $\Diamond\mathbb{P}$, small timeout values can be used initially. The values can be subsequently increased

(exponentially) if they are proven to be too short and to generate false positives. Doing so helps to provide eventual accuracy.

Implementing $\Diamond\mathbb{P}$ in this way, however, may leads to large timeout values. A faulty server can take advantage of large timeout value to temporarily delay the commitment of learned requests. Though not violating liveness, this make it possible for malicious servers to mount a performance attack [6] that can result in a significant increase of request latency. So, we use stricter criteria for detecting failures in our protocol. This may leads to correct but slow servers being permanently falsely suspected. To compensate that we allow a falsely suspected server to forward its requests to another server, much like Rule RL.5 of protocol $\mathscr{R}$.

Rule R.5: If server p is being revoked, upon receiving a request v from its client or upon learning *no-op* for an instance to which it have previously proposed client request v, instead of suggesting or re-suggesting v, p forward v to another server r that p believes is correct. When receiving v, r treats it the same as a client request.

Rule R.5 implies that when the forwarder is later suspected to be faulty, a server may re-forward requests if needed.

Theorem 6.9: Assuming Ω, $\mathscr{Q}$ with Rule R.1 – R.5 implements replicated state machines.

Proof. We already know that $\mathscr{Q}$ always satisfies R.2 (Agreement), R.3 (Integrity) and R.4 (Total Order) from Lemma 4.6. So, we only need to prove R.1 (Validity). From the proof of Lemma 6.7, we know that only the requests sent to falsely suspected servers are not live. Rule R.5 allow a falsely suspected server p to forward request to another server r, and r will suggest the request. Since we assume Ω, p can eventually find a correct server r to which to forward requests. Since any request suggested by a correct server is guaranteed to commit eventually, any request submitted to p is also guaranteed to be live. So, $\mathscr{Q}$ with Rule R.1 – R.5 satisfies Validity and hence implements replicated state machines correctly. □

6.5.3 Optimizations

This message complexity of the basic protocol is high due to the broadcast of SKIP, ACCEPT, SU-RELAY, and SK-RELAY messages. The ACCEPT messages are broadcast by design, the other three class of messages can be piggybacked on other messages to reduce message complexity.

Recall that Mencicus reduces message complexity by piggybacking SKIP messages on subsequent ACCEPT and future SUGGEST messages. Mencius must rely on future SUGGEST messages because it does not broadcast ACCEPT message due to its goal to improvement throughput. RAM, with its focus on reducing latency, broadcast ACCEPT messages. Doing so is useful not only for dealing with faulty servers that omit messages to a fraction of the servers (see section 6.4), but also for making piggybacking eaiser: SKIP and SU-RELAY are always generated at the same time as ACCEPT messages.

Optimization R.1: SKIP and SU-RELAY are always piggybacked on ACCEPT messages.

Note that Mencius also overloads the semantics of ACCEPT and SUGGEST messages to further reduce the number of fields required in the piggybacked messages. Because of the crash-failure semantics, all fields of the skip message can be inferred from the overloaded ACCEPT or SUGGEST messages, and so no additional field is added to the piggybacked messages. This technique is, however, not effective with Optimization R.1. This is due to the use of digital signature under the Byzantine failure model: the signature fields must present in the piggybacked messages.

SK-RELAY messages can also be piggybacked or combined using the following optimization:

Optimization R.2: SK-RELAY are buffered to be piggybacked on other messages. A protocol parameter ζ specifies the maximum number of seconds an SK-RELAY message can be buffered. Once ζ seconds is reached, all outstanding SK-RELAY message are sent out immediately.

RAM can afford the extra latency introduced by Optimization R.2 because SK-RELAY message are not time critical. For example, a simple way to select ζ is to set it to $1/10$ of the estimated one-way delay. In our evaluation environment, one-way delay is

50 ms, and so ζ is set to 5 ms, resulting in at most 200 combined SK-RELAY messages being sent out every second. This helps to reduce the message complexity at the cost of relatively low extra latency.

Lemma 6.10: Protocol $\mathscr{2}$ with Optimization R.1 and R.2 implements replicated state machines correctly.

Proof. Optimization R.1 merely combines multiple messages that $\mathscr{2}$ would have sent separately into one message. Optimization R.2 delays the SK-RELAY messages for bounded time. Doing this clearly does not violate any of the safety properties. It does not violate any of the liveness properties either because $\mathscr{2}$ only relies on the eventual delivery of messages for liveness: up to ζ seconds are added to the message latency. □

Like $\mathscr{R}$, $\mathscr{2}$ can also reduce the overhead of revocation by revoking suspected servers in large block. This results in the following optimization:

Optimization R.3: Let q be server that $f+\alpha$ servers suspects has failed, and let p be a non-faulty server that is elected to revoke q. 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 $C_q < I_p+\beta$.

Corollary 6.11: Protocol $\mathscr{2}$ with Optimization M.3 implements replicated state machines correctly.

Proof. From Lemma 4.13 □

Definition 6.3 (RAM): RAM is $\mathscr{2}$ (Rule R.1 – R.5) combined with Optimizations R.1 – R.3.

Theorem 6.12: RAM implements replicated state machines correctly.

Proof. From Lemma 6.10 and Corollary 6.11. □

6.6 Discourage uncivil rational behavior

We have explained the basic structure of RAM. In this section, we discuss how RAM discourages uncivil rational behavior. The root cause of rational behavior is faulty

and/or selfish administrators who exploit the protocol to improve local utility possibly at the cost of the utility of remote sites. Such behavior can be discouraged by reinforcing the protocol so that any divergence from the correct behavior only increase the chance of lowing local utility. The core mechanism for RAM to accomplish this revocation.

With the rotating leader design of RAM, all servers have the opportunity to make requests directly for their clients. Revocation means that being a coordinator is a privilege rather than a right. Since RAM, by design, prevents a server from bypassing revocation, a revoked server can only forward its clients' requests to other servers. Forwarding has two obvious drawbacks: it adds to the latency of client requests and it requires the revoked server to find a correct server to be the forwarder.

Thus, it is in the best interest of any rational server to avoid being revoked. In fact, in some cases it would make sense to sacrifice a bit on latency for local requests to avoid the higher latency that revocation would induce. Thus, revocation constitutes a fundamental tool in RAM to discourage uncivil rational behavior. The rest of the section explains RAM's revocation policy as well as the implications of using revocation.

6.6.1 Failure detection under network jitter

RAM is designed to revoke servers that are suspected to be faulty. Any practical implementation of a failure detector is likely to produce false suspicions due network jitter. But, a falsely suspected (and therefore revoked) server can not recover quickly. This means – not surprisingly – that, in addition to eventual accuracy, the underlying failure detector should avoid false positives. A simple approach to do this is to have the failure detector report three possible values: *correct*, *suspicious*, or *faulty*.

Correct: A server is detected as correct if it is timely and no misbehavior is detected.

Faulty: A server is faulty if definite misbehavior is detected, *e.g.*, proposing inconsistent requests to the same consensus instance because of a faulty A2M. Upon detecting faulty behavior, a correct server raises a verifiable suspicion (discussed in section 6.6.4) for this class of failures, and a faulty server is repeatedly revoked until it has been repaired via human intervention.

Suspicious: Finally, a server is considered *suspicious* if it is not acting fairly or in a timely fashion. Examples of such behaviors are failing to batch when it has a 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 environmental issues like network jitter, or if the actions are deliberate. Upon detecting such a behavior, a correct server raises an unverifiable suspicion (see section 6.6.3).

The protocol has a parameter α such that $f + \alpha$ unverifiable suspicions are required to start revocation. Ideally, a practical failure detection implementation has a low false positive rate resulting from network jitter. A larger value of 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. As a consequence, a rational server should avoid being unnecessarily suspected, since it would take fewer servers noting jitter for it to be revoked.

Since revocation is used as a punishment for uncivil behavior, the time period associated with those revocation triggered by unverifiable suspicion should, at least initially, be small because of occasional false suspicions arising from bad environmental conditions. If a server continues to be suspected frequently enough, then the time period can be increased.

6.6.2 Turnaround time advertisement

At the core of RAM's failure detection is a *turnaround time advertisement* mechanism. Similar mechanism has been used by others to select a fast leader for BFT protocols [6]. Under this mechanism, a server p is required to periodically advertise a *turnaround time* $L_{p \to q}$ between itself and each other server q. This value is an upper bound on the amount of time q will need to wait to get a response from p concerning a message that requires immediate response (such as a SUGGEST message). Missing this deadline may cause q to suspect p.

To give this kind of advertisement, p needs to predict round-trip latency. Predicting round-trip latency for wide-area network has been an active research topic in the

Grid community and promising results have been obtained [12, 45, 59]. Once signed and advertised, these promises are collected and fed into each server's failure detector module as dynamic input to produce a list of up to f suspected servers. As we will discuss later it is in the best interest of any rational server to provide a promise that is as accurate as possible.

6.6.3 Unverifiable suspicion

Suspicion in RAM can be roughly categorized into two classes: *verifiable* and *unverifiable* suspicions. A server p raises an *unverifiable suspicion* against server q based on information that is collected locally and can not be verified globally. For example, p may suspect q because q is slow, but p cannot prove this to a third party; perhaps q is timely and p is attacking q deliberately. In the following, we give a list of conditions that lead to an unverifiable suspicion. This list covers some important and generic conditions for unverifiable suspicions, but it is not meant to be comprehensive. Specific environments or system performance requirement will introduce other conditions as well.

Untimely behavior

When a server p sends a message (such as SUGGEST) to q that requires q's immediately response, p raises an unverifiable suspicion against q if q's response time does not meet q's latency promise $L_{q \to p}$. This revocation condition gives incentive for a server to not to advertise latency promises that it cannot deliver, as well as to get reply to other servers as soon as possible to avoid untimely behaviors caused by unexpectedly large network jitter.

We have discussed that network jitter makes it difficult to distinguish a deliberately slow server and a server whose messages happen to have jitter. It is interesting to note that by using the turn-around time and revocation mechanism, network jitter is turned from something a uncivil rational server can take advantage of into something encourage rational server to act civilly.

Flooding

One type of uncivil behavior is for a server to flood the system with local requests when it is experiencing high loads from its client. This behavior can potentially improve throughput and latency for its local clients at the risk of overloading the system and reducing the over all utility of the system as a whole. To discourage such a behavior, a server is required to either batch or traffic shape its local requests so that it does not exceed a predefined throughput threshold. Failing to comply results in suspicion and hence revocation.

A simple strategy to select throughput threshold is to set it to 80% of throughput a server can handle. For example, each server in our prototype implementation can handle approximate 200 ops (operations per second) (see section 6.8.4) – largely independent of the request size. In a system with a total of four servers, setting the threshold to 160 ops would allow a total of 640 ops before batching kicks in. The extra latency caused by batching is no larger than $1/160$ second or 6.25 ms, which is quite small compared to the 100 ms round-trip latency in our experiments.

Forwarding omission

When a server p is under revocation, it cannot directly propose requests; it has to forward its local requests to another server q. To prevent p from overloading q, RAM allows q to buffer such forwarded requests for a short period of time ρ to seek for opportunities to batch requets. An uncivil server may pretend it has not received the forwarded message and delay proposing them for a long period of time. To mediate this behavior, p signs and broadcasts the request to all servers with a designation that q is to propose it. Upon receiving the request from p, a third party r may also relay the request to q, expecting to receive a SUGGEST from q within $\rho + L_{q \to r}$. r suspects q if such an expectation is not met. A server r decides to do this based on a random variable, set so the expected number of servers forwarding the request is larger than f. While this mechanism does not completely prevent q from sitting on forwarded requests, it does put a upper bound on how long it can do so.

Note that ρ should be set to be no less than the inverse of a server's throughput threshold. For example, if the threshold is set to 160 ops, ρ should be at least 6.25 ms

to make sure p cannot force q to exceed the throughput threshold.

Finally p is free to designate whichever server to propose its requests; it can therefore try multiple servers and decide on the one that gives the best response time, making forwarding omission less a threat.

Handling forwarded requests requires a server to do extra work: (1) the server needs to buffer and handles batching; and (2) A2M needs to sign the batched and hence larger requests. One potential problem is handling forward can hurt the ability of a server to handle requests from its own clients. The cost of buffering and batching is relatively small and negligible comparing to the A2M signing cost, and so, is not a issue. From the evaluation result in section 6.8.4, we can see that the number of requests the A2M thread can sign is roughly the same regardless the size of the requests. So handling larger requests would not significantly hurt the ability of a server to handle requests for its own clients.

Reciprocal suspicion

In section 6.6.4 we define a policy that determines which servers should be suspected because they are slow. This policy suspects no more than f servers. Because of this, a uncivil rational server may attempt to reduce the chances of itself being suspected by increasing the chances of other servers being suspected. A rational server can do this by simply suspecting other servers for no valid reason in hopes that network jitter will result in enough suspicions. To discourage this behavior, we allow a server to raise a reciprocal suspicion. For example, q can raise a reciprocal suspicion against p should p suspect q is untimely but q has been timely and has not recently experience network jitter between p and q. The rationale behind a reciprocal suspicion is that suspecting a correct server is itself a suspicious behavior since there is nothing preventing a faulty server from suspecting any server. Suspicion retaliation gives incentive for rational servers not to abuse suspicions, since they come at a cost.

6.6.4 Verifiable suspicion

Verifiable suspicions are those that can be verified by any server. A correct server q becomes suspicious of r upon receiving a verifiable suspicion against r, and so one

verifiable suspicion from server p against server r is sufficient to initiate revocation against r. Most verifiable suspicions are raised as a result of definite Byzantine behavior. For example, if p's A2M is compromised, p may tell server q that it has suggested $v \neq$ *no-op* to instance i and tell server r it has skipped instance i. By exchanging SU-RELAY and SK-RELAY messages, q and r can easily detect this failure and raise a verifiable suspicion. Verifiable suspicions can also be raised when a server violates the definition of simple consensus, *i.e.*, when a follower uses its A2M to suggest $v \neq$ *no-op*. The rest of the section discusses another class of verifiable suspicions and describe how they are used to discourage a server from advertising inaccurate turnaround times.

We noted in section 6.6 that it is in the best interest of a rational server to advertise a turn-around time that it can deliver. Without further restriction, this gives incentive to rational servers to advertise promises that are higher than what they can actually deliver; doing so makes it easier to meet their promise and avoid being revoked. The drawback of this is that uncivil rational servers could use this additional slack time to prioritize its own requests without risking itself being suspected and revoked. RAM uses the follow mechanism to discourage this behavior by increasing a server's probability of being revoked if it reports an abnormally large turn-around time.

The combined turn-around promises can be represented as a graph with vertices as servers and promises as edges. Each server r collects the advertised turn-around time between each pair of servers, *i.e.*, $L_{p\to q}$ and $L_{q\to p}$ are collected for each pair of servers p and q. Let $L_{p\leftrightarrow q} = \max\{L_{p\to q}, L_{q\to p}\}$. r constructs a fully connected graph G of the servers using the values of $L_{p\leftrightarrow q}$. Note that the larger of $L_{p\to q}$ and $L_{q\to p}$ is used to construct G because, were the lower of the two is used, it would allow p to provide a high turn-around time promise without being observed in G as long as q has a lower promise.

Let Π be the set of all servers. We define a utility function $U(G,S,p)$ that, given G and a set S ($S \subseteq \Pi$) of revoked servers, estimates the utility of server p. Then a deterministic policy P is used to decide if a server q is too slow, *i.e.*, if revoking q would result in a better global utility. If so, then server q is suspected and revoked. One requirement of $U(G,S,p)$ and P is that increasing $L_{p\leftrightarrow q}$ increases the chance of p and q being considered slow; a server that reports a large turnaround promise is consequently

more likely to be revoked.

The exact utility function $U(G,S,p)$ and policy P to be used depends on the specific system and should try to optimize the global utility function of the system. While the exact details on selecting U and P are outside the scope of this dissertation, we present the following examples:

Definition 6.4 (Utility function U_1): We define $U_1(G,S,p)$ to estimate the expected latency of server p: Note that the higher $U_1(G,S,p)$ is, the lower utility server p is expect to have.

$$U_1 = \begin{cases} \max\{L_{p\leftrightarrow q} : q \notin S\}, & \text{if } p \notin S \\ \min\{\frac{1}{2}L_{p\leftrightarrow q} + \frac{1}{2}\max\{L_{q\leftrightarrow r} + L_{r\leftrightarrow q} : r \notin S\}\}, & \text{if } p \in S \end{cases}$$

We used the above definition for the following reasons:

- When p is not revoked, the expected latency of p is the highest expect round-trip latency between p and any server q that is not revoked. This is because p needs to receive an acknowledgment from all the servers that are not currently being revoked to commit its proposal.

- When p is revoked, p uses a server q that is not currently being revoked to forward its requests. The expected latency of p via q consists of three parts: (1) one way delay between from p to q for p to forward the request to q; (2) one way delay from p to all non-revoked servers for q to propose the request; and (3) one way delay from all non-revoked servers to q for sending the ACCEPT and SKIP messages. After enough ACCEPT and SKIP messages are collected, p and execute the request and send a reply to the client via local communication. Assuming link latencies are symmetric, the first part is then estimated as $\frac{1}{2}L_{p\leftrightarrow q}$. The second and third parts of the latency need to be considered together. They are the highest one-latency from q to p via a non-revoked server r, since p needs to know that all non-revoked server has skipped before committing the request. So, they are estimated as $\frac{1}{2}\max\{L_{q\leftrightarrow r} + L_{r\leftrightarrow q}\}$. We then estimate the expected latency of p as the lowest latency possible using any of the servers that is not being revoked[2].

[2]By using the lowest latency possible, we do not consider faulty servers. This gives a low bound of

We have defined $U_1(G,S,p)$ to estimate the expected latency of server p when the servers in S are revoked. Given G, we now define policy P_1 that uses U_1 to decide whether a set of servers S is too slow, *i.e.*, whether revoking the servers in S would result in better utility for the system as a whole. First, we introduce two supportive definitions:

Definition 6.5 (Expected bottom-line latency): Given a set of non-revoked servers T (*i.e.*, $T \cap S = \phi$), we define the *expected bottom-line latency* of T as: $\widehat{U_1}(G,S,T) = \max\{U_1(G,S,p) : p \in T\}$. This is the highest expected latency of all servers in T.

Definition 6.6 (Valid revocation set): We say a set of servers S is a *valid revocation set* under policy P_1 iff $|S| \leq f \wedge \frac{3}{2}\widehat{U_1}(G,S,\Pi \setminus S) \leq \widehat{U_1}(G,\phi,\Pi \setminus S)$, *i.e.*, a set of no more than f servers are considered as a valid revocation set if revoking the servers in S would reduce the bottom-line latency of the rest of the servers to reduce by at least $1/3$.

The threshold $1/3$ is somewhat arbitrary. The goal is to avoid making revocation too aggressive: servers are only revoked if they are consider as too slow and revoking them would benefit the rest of the servers enough to outweight the increased latency that would occur for servers in S.

Definition 6.7 (Policy P_1): Policy P_1 uses Algorithm A_1 to compute a minimal-sized valid revocation set that would result in the best possible bottom-line latency for the servers that are not revoked. A_1 returns either ϕ or a set of servers sets, all of the same size. With P_1, no server is revoked if A_1 returns ϕ; all servers in the set are revoked if A_1 returns a single set of servers; when A_1 returns multiple sets, one set is chosen deterministically and those servers are revoked.

It is straightforward to see that U_1 and P_1 meet the requirement that increasing $L_{p\leftrightarrow q}$ increases the chance of p and q of being considered as too slow, *i.e.*, p or q could be revoked if all the other servers advertise low latencies among themselves.

Therefore, it is in the best interest of a rational server p to advertise $L_{p\rightarrow q}$ as low as possible – advertising unnecessarily high $L_{p\rightarrow q}$ results in unnecessarily high $L_{p\leftrightarrow q}$, which in turn unnecessarily increases the chance of p being revoked. We have already

the utility of server p. If it is desired, one can always estimate the utility of a server by using the $(f+1)^{th}$ smallest latency, which guarantees a correct server is used for forwarding. Doing so gives an upper bound of the utility of server p.

1: **procedure** $A_1(G)$
2: $SS \leftarrow \{S : S \text{ is a valid revocation set in } G\}$
3: **if** $SS = \phi$ **then**
4: **return** ϕ
5: **end if**
6: $m \leftarrow \min\{\widehat{U_1}(G,S,\Pi \setminus S) : S \in SS\}$
7: $SM \leftarrow \{S : \widehat{U_1}(G,S,\Pi \setminus S) = m \wedge S \in SS\}$
8: $ms \leftarrow \min\{|S| : S \in SM\}$
9: **return** $\{S : |S| = ms \wedge S \in SM\}$
10: **end procedure**

Figure 6.4: Algorithm A_1

discussed that p has incentive to not advertise $L_{p\to q}$ to be lower than that it can deliver. Combining the two results, we have that a rational server p has the incentive to advertise $L_{p\to q}$ to be as accurate as possible.

We introduced U_1 and P_1 here to demonstrate how to use them to discourage unnecessary high latency promise advertisement. However, it is likely more sophisticated policy should be considered for practical deployment. Also the complexity of algorithm A_1 used by P_1 is exponential in n, the total number of servers. While we do not expect the complexity to be an issue for small n, should it becomes one, an approximate algorithm can be used instead. For example, we can start with an empty revocation set and adds the slowest server to the revocation set until adding such a server would result in an invalid revocation set under P_1 or the size the revocation set reaches f.

We also report a latency upper bound obtained using U_1 and P_1 in section 6.7.4 and evaluate the latency of the servers under U_1 and P_1 in 6.8.5.

6.7 Combating irrational behavior

In the previous section, we have discussed how to use incentive to discourage uncivil rational behavior. Irrational behavior is less restricting – a compromised server may become untrustworthy to its local client or cooperate with other compromised servers

in an attempt to mislead other servers to enter into an unsafe state. Although it is impossible to implement a perfect failure detector for Byzantine failures (or even for crash failures [17]), we wish to expose as many Byzantine behaviors as possible so that correct servers can exclude faulty servers.

This section discusses the Byzantine failures that attack system assumptions: trustworthy A2Ms and the MSD model (clients trust the local server). Other Byzantine failures pose no risk to the safety of the system assuming that no more than f servers are faulty. This is because Coordinated Byzantine Paxos is derived from PBFT, which masks up to f Byzantine failures. Detecting and removing Byzantine servers, however, is a good idea – doing so is a well-known technique for improving fault coverage.

When detection of faulty behavior is impossible, RAM resorts to damage control methods that prevent a significant drop of system utility: RAM impose an upper bound on the extra latency a Byzantine servers can cause regardless their faulty behavior is detectable or not. We discuss this in section 6.7.4.

6.7.1 Compromised A2M

We have so far assumed A2M is a trustworthy third party and used it to remove one wide-area communication step. If a server p and its local A2M are both compromised, then, for a consensus instance i, it is possible for p to suggest a value $v \neq$ *no-op* to server q and propose *no-op* to server r. Server r might go to an unsafe state by immediately learning *no-op* for instance i upon receipt of the proposal. Note that q does not enter an unsafe state because it still requires a quorum of the servers to accept the request for it to learn the outcome. As discussed in section 6.6.4, this behavior can be detected by exchanging SK-RELAY and SU-RELAY messages. Once detected, server r and its clients could roll back their states and recover using some rollback methods such as transactions [27]. Doing so also requires that a client does not send output to users until it knows the state machine command has committed [64]. Note that even in face of faulty A2M, r can confirm *no-op* is the consensus outcome for instance i once it receives additional $2f$ matching SK-RELAY messages; doing this only takes one extra wide-area communication step. Such a short speculative time frame enables an efficient management of speculative states – for example, not allowing output commit – and make it

possible to commit stable states quickly.

The way RAM deals with faulty A2M can also be used to enable RAM when A2M hardware is not available. In this scenario, a RAM server simply signs a proposal when suggesting values or skipping turns. The signed proposal is treated as if it is signed by the A2M. If inconsistent proposals are later discovered, servers rollback their states to recover. Essentially, we use untrustworthy software to simulate hardware A2M and rely on recovery mechanism for correctness. Doing so does not introduce new mechanism but does make RAM more vulnerable when one or more servers are compromised. Fortunately, such faulty behavior is easily detectable and is verifiable at all servers. Once detected, the faulty server is excluded from further action to prevent further disruption of the replicated state machine.

6.7.2 Untrustworthy local servers

So far we have assumed the MSD model, in which a client trusts its local server and consequently learns the outcome of its requests as soon as it receives a reply from its local server. Trusting its local server saves one wide-area delay to a client for every request. We have also assumed in the MSD model that the servers have incentives to reduce latency for its clients, and therefore propose the local requests and send correct reply back to a client as soon as possible. A compromised server can violate this assumption by, for example, delaying proposals or sending incorrect replies.

RAM uses the following technique to expose such servers. For a server that deliberately delays the processing of local requests, a RAM client raises an alert against its local server if the request latency is unacceptably high. For a server that sends fabricated replies, a server is required to collect replies from other servers and forward them to the client using O-REPLY messages [3]. If the client does not get f additional O-REPLY messages that match the original reply within a reasonable time, it raises an alert against the local server.

Note that raising an alert against a local server does not lead to the server being revoked. Instead, it alerts local administrator that either (1) the client is compromised,

[3] Since it is a client that eventually verify the O-REPLY messages, when MACs are used to authenticate messages, a remote server authenticates the reply sent to the local server using the shared secret key between the client and the remote server

(2) the server is compromised, or (3) the site is experiencing unexpected network problems. Since the client and the server are under the same administrative domain, administrators can use support systems (*e.g.*, intrusion detection systems [55,66]) and forensic tools [51] to determine the culprit. If neither the server nor the client is deemed as faulty, then the alert means this is a good time for local administrators to investigate what is the cause of the underlying network problem.

6.7.3 Ignoring revocation

One possible Byzantine behavior is for a server to ignore its own revocation. A server q can pretend it is not aware of its own revocation and continue to suggest requests even though q knows other servers would ignore these suggestions. Doing so does not help q's latency but can slightly increase the load on the system. We use the following mechanism to combat this behavior. Upon first learning that q is being revoked, a server p sends a revocation certificate to q and expect q to confirm within time $\gamma \cdot L_{q \to p}$ for some protocol parameter γ. Failing to confirm in time or suggesting requests after confirmation will result q being more seriously suspected and revoked for longer period of time. γ should be chosen to allow reasonable network jitter so that with high probability q is able to confirm with in $\gamma \cdot L_{q \to p}$ seconds.

6.7.4 Latency upper bound

In this section we explain why a Byzantine server can only cause a limited increase in the latency to requests proposed by other servers without the risk of being revoked. We consider a period of synchrony in which communication is timely and the revoked set is stable. We assume this because it is well known that consensus is impossible during period of asynchrony [26], and so no upper bound can be obtained. We first obtain this upper bound in term of number of communication steps. We then apply utility function U_1 and policy P_1 to obtain an upper bound in term of the communication delay between correct servers. For simplicity, we ignore the latency introduced by delayed commit, which can add up to one more one-way message delay when concurrent requests are proposed by the servers.

Upper-bound on communication steps

A RAM server participates in the state machine in two ways: it acts both as the *proposer* for instances coordinated by itself and as an *acceptor* for instances coordinated by other servers. Up to f Byzantine server can only cause a limited increase in latency for servers that are not currently under revocation because:

1. As an acceptor, a Byzantine server can not keep requests proposed by a correct server from being chosen: Coordinated Byzantine Paxos 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 that server: $f+\alpha$ servers are required to revoke a server.

3. As a proposer, a Byzantine server can not introduce 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 or suggest values for their 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 $(f+\alpha-1)^{th}$ SU-RELAY messages. Doing so adds one wide-area message delay at most. After receiving the $(f+\alpha-1)^{th}$ SU-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, which ensures that the extra latency is at most one wide-area message delay.

Thus, the total extra delay a Byzantine server can induce to a server that is not currently under revocation is two wide-area message delays, *i.e.*, four steps is the upper bound of the total delay.

For a server r that is currently being revoked, one more delay is added to the upper bound, *i.e.*, five steps is the total, if r can find a correct server to forward requests to. Otherwise, yet another step is added to the upper bound, bring the total to six steps, because a Byzantine server can refuse to forward a request for up to one step without being revoked (discussed in section 6.6.3).

We summarize the above result with the following theorem.

Theorem 6.13: The extra delay L in communication steps that Byzantine servers can induce to a server p is:

$$L = \begin{cases} 2, & \text{if } p \text{ is not being revoked} \\ 3, & \text{if } p \text{ is being revoked and a correct server is used for forwarding} \\ 4, & \text{if } p \text{ is being revoked and a faulty server is used for forwarding} \end{cases}$$

Upper-bound on communication delays

In this section, we apply the utility function U_1 and policy P_1 that we discussed in Theorem 6.13 to the upper-bound obtained in the previous section. First, we discuss the model and introduce a few supportive definitions:

Let $D_{p\to q}$ be the one-way delay from server p to q. For simplicity, we consider a simple model where $D_{p\to q} = D_{q\to p}$, *i.e.*, the link latencies are symmetric between each pair of servers.

Definition 6.8 (Correct quorum): Let $\mathscr{C}(|\mathscr{C}| \geq 2f+1)$ be the set of all correct servers. A set of servers C is called *a quorum of correct servers* iff $C \subseteq \mathscr{C} \wedge |C| = 2f+1$.

Definition 6.9 (Intra-quorum latency): The *intra-quorum latency* of a quorum of correct servers is defined as $t_C = \max\{D_{p\to q} : p,q \in C\}$.

Definition 6.10 (Fastest correct quorum): A quorum of correct server $\hat{C}$ is called a *fastest correct quorum* if $\hat{C}$ has the lowest intra-quorum latency among all possible correct quorums.

Note that, it is possible to have more than one fastest correct quorums, for example, if all the servers are correct and have the same inter-server latency, then any $2f+1$ servers make up a fastest correct quorum.

Lemma 6.14: Let $\hat{C}$ be a fastest correct quorum and denote $t_{\hat{C}}$ with τ. A server p will be revoked under U_1 and P_1, if $\forall q \in \hat{C} : D_{p\to q} \geq 1.5\tau$.

Proof. Let Π be the set of all servers, it is not difficult to verify that $\Pi \setminus \hat{C}$ is a valid revocation set: $|\Pi \setminus \hat{C}| = f$, $\widehat{U_1}(G, \phi, \hat{C}) \geq 3\tau$, and $\widehat{U_1}(G, \Pi \setminus \hat{C}, \hat{C}) = 2\tau$.

Now assume p is not revoked under U_1 and P_1. There must exist a revocation set S such that $p \notin S \wedge \widehat{U_1}(G, \Pi \setminus S, S) \leq 2\tau$. Thus, $\forall s,t \in \Pi \setminus S : D_{s \to t} \leq \tau$. This is impossible because at least $f+1$ servers in $\Pi \setminus S$ are also in $\hat{C}$, the one-way latency between p and these $f+1$ servers is at least 1.5τ. □

If a server p is revoked, p need to forward its requests to another server. By assumption, the set of revoked servers does not change during a period of synchrony. p knows which servers are revoked: revocation certificates are broadcast and *no-op* is chosen for turns coordinated by the revoked servers. To avoid a forwarding chain that adds further latency to its request, p only forwards requests to those are not currently being revoked.

Theorem 6.15: Let $\hat{C}$ be a fastest correct quorum and denote $t_{\hat{C}}$ with τ. Let p be a correct server and $l_p = \max\{D_{p \to q} : q \in \hat{C}\}$. Let L_p be the latency at server p, we obtain the following upper bound:

1. If p is not being revoked, $L_p \leq 3\tau + 2l_p$;
2. If p is being revoked and a correct server in $\hat{C}$ is used for forwarding, then $L_p \leq 4\tau + 2l_p$;
3. If p is revoked and a server not in $\hat{C}$ is used for forwarding, then $L_p \leq 5.5\tau + 2l_p$.

Proof. If p is not being revoked, p's proposal commits when p receives the acknowledgement to its SUGGEST message from all servers that are not being revoked. The relay mechanism in RAM ensures that, $l_p + 1.5\tau$ seconds after p suggests its request, all unrevoked servers receives at least $2f+1$ copies of the SUGGEST message from the servers in $\hat{C}$. The unrevoked servers, correct or Byzantine, must have generated acknowledgement and sent it to one of the servers in $\hat{C}$ – a correct server does so when it receives the first copy of the SUGGEST message; a Byzantine server does so upon receiving the $(f+\alpha)^{th}$ copy to avoid being suspected by $f+\alpha$ servers. The acknowledgement takes at most $1.5\tau + l_p$ to reach p: at most 1.5τ to reach a correct server c in $\hat{C}$ and at most l_p from c to p. Therefore, p can commit its proposal by $3\tau + 2l_p$.

Similarly, if p is being revoked and a correct server c in $\hat{C}$ is used for forwarding. p's requests arrives at c in l_p seconds. p immediately propose the request. a Byzantine server does not have to acknowledge until it receives the $(f+\alpha-1)^{th}$ SU-RELAY message, which takes at most 2.5τ. The acknowledgement then takes at most $1.5\tau + l_p$ to arrive at p. Therefore, the upper bound of this case is $4\tau + 2l_p$.

Finally, if p is revoked and a server q not in $\hat{C}$ is used for forwarding. By $1.5\tau + l_p$ seconds, q must have received at least $2f+1$ copies of p's request from the servers in $\hat{C}$. By this time, q, correct or Byzantine, must have proposed the forwarded request. This introduce an additional 1.5τ seconds delay, bringing the upper bound to a total of $5.5\tau + 2l_p$. □

6.8 Evaluation

In this section we report the results of experiments in the DETER testbed [10] to evaluate RAM. We also compare RAM's performance with PBFT's baseline performance obtained from other sources [14, 33]. First we summarize our prototype implementation and explains our experimental setup. We then evaluate RAM's latency and throughput. Finally, we presents a setup in which revocation is in action as a first step to understand the implication of revocation.

6.8.1 Prototype

We implemented RAM in Java 1.6 and used Sun's implementation of cryptographic primitives. We used TCP as the transport protocol and disabled Nagle's algorithm [49] to reduce latency. Out-of-order commit was implemented as an option that can be dynamically turn on or off.

Like Mencius, RAM implements an API with three calls: The application calls PROPOSE(v) to issue a request, and the state machine calls back the application using ONCOMMIT(v) once the request is ready to commit. We use a second callback ISCOMMUTE(u,v) to ask the application if two requests are commutable when out-of-order commit is enabled.

For evaluation, we used the read/write register service that the evaluation of

Mencius used. Such a service is attractive because it is simple and allows out-of-order commit operations. In more detail, this service implements a total of κ registers, a read command, 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 payload (this payload is ignored for read operations and is only used to simulate read operations of specific size).

6.8.2 Experimental setup

Unless otherwise stated, we use a four-site fully-connected topology. We set up a virtual link between each pair of sites, and each link had, by default, 50 ms one-way delay and 20 Mbps bandwidth. Due to hardware constraints, we used one node for each site – clients and the server of the same site are running on the same machine. The configuration of each node is the following: 2.13 GHz Quad-Xeon PC, 2.0 GB memory, running Ubuntu 8.

Table 6.2: Micro benchmark on security primitives

Hash function	MD5		SHA1	
Data size (byte)	64	2048	64	2048
Hash MAC ($\times 10^4$ops)	513	311	82.3	48.1
RSA Sign (ops)	221	219	222	221
RSA Verification (ops)	4210	4195	4035	3879

A micro benchmark of the throughput of the primitives measured in ops (operation per second) is given in table 6.2. 1024-bit keys were used for all RSA operations. As expected, Hash MAC is significantly faster than RSA. In addition, the throughput of RSA operations are nearly independent of the hash function used and the data length. Since PBFT uses MD5, we only present RAM's performance measured operations that use MD5 as the hash function. A multiple thread implementation was used to take advantage of the availability of multi-core machines. A2M was implemented in a separated thread and was the single bottleneck of the system as we show later.

We verified that running clients and the server on the same machine had minimal impact on the performance of the servers – each client used less than 1% CPU, whereas

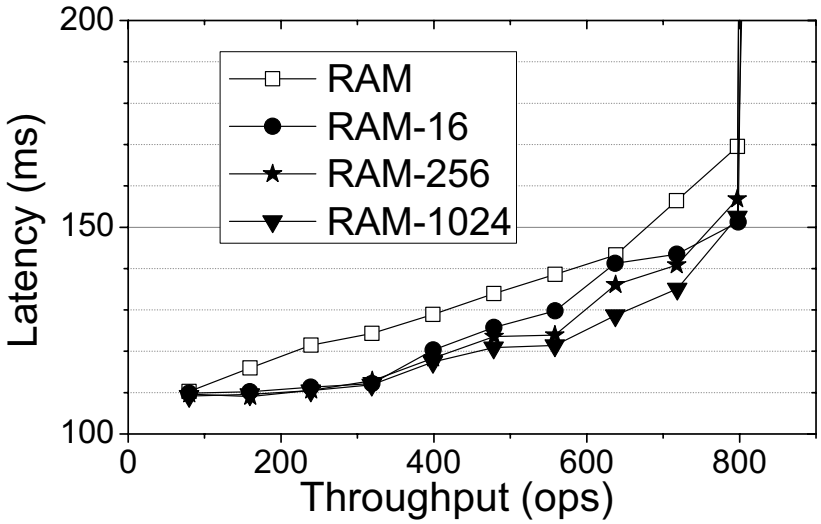


Figure 6.5: Commit latency vs. offered client load

servers used 170%[4] CPU at peak utilization. An A2M signature cache was implemented at the servers to more efficiently verify SU-RELAY and SK-RELAY. Using a cache reduced the CPU utilization from 170% to 125% at the peak. However, it did not improve throughput significantly, since the cache did not improve the system bottleneck, which was the A2M signing thread.

Clients submit operations at a fixed rate. Each operation is either a read or a write operation, selected randomly, to a register uniformly chosen from the set of registers with a fixed payload of ρ bytes. For the following experiments, we did not enable local processing of read operations, so all operations go through the replication protocol.

6.8.3 Latency

In this section, we measure RAM's latency under different offered client loads. `RAM` denotes RAM with out-of-order commit disabled, and `RAM-`κ denotes RAM with out-of-order commit enabled and κ registers. For example, `RAM-16` implements 16

[4]The CPU utilization is over 100% due to the use of multi-core machines.

registers. As we have discussed in section 5.5.5 having more registers in the service implies a higher chance that two commands are commutable, which causes the out-of-order commit technique to be exercised more often.

Four RAM variants were used in the experiment: `RAM`, `RAM-16`, `RAM-256`, and `RAM-1024`. Figure 6.5 shows the average delay of a 10-second period for each data point.

The expected minimal delay for RAM is 109 ms, *i.e.*, the round-trip delay (100 ms) plus the delay of two RSA sign operations, one for signing SUGGEST and one for signing SKIP (4.5 ms each). Indeed, all four variants started with a delay of approximately 110 ms. For `RAM`, latency gradually increases to about 160 ms and ramps up before reaching the maximum throughput at around 800 ops. This extra delay was caused by the delayed-commit problem, which adds up to one way delay (50 ms) depending on the level of contention – higher load causes more contention, and increases latency. All three out-of-order enabled variants showed noticeable latency improvement over `RAM`. `RAM-1024` achieves the maximal latency improvement: approximate 15%. This value is significantly smaller than the 35% that Mencius achieves, but that is due to RAM using broadcast to reduce the upper-bound of delayed commit from two steps to one step. Up to 320 ops, all three variants show latency close to the minimal. As the throughput ramps up, the difference between the variants with out-of-order commit enabled increased: `RAM-1024` had better latency than `RAM-256`, which in turn had better latency than `RAM-16`. As expected, all three variants always have better latency than `RAM`.

For comparison, from a communication pattern analysis [14], PBFT would have at least 200 ms delay at the primary and at least 250 ms at the followers, resulting in an average of 237.5 ms. This is significantly higher than that of any of the four RAM variants.

6.8.4 Throughput

This section evaluates RAM's throughput, which was measured by running a large number of clients to overload the servers. The result is shown in figure 6.6. Note that error bars are not shown in the figure as they are small. Table 6.2 shows that the

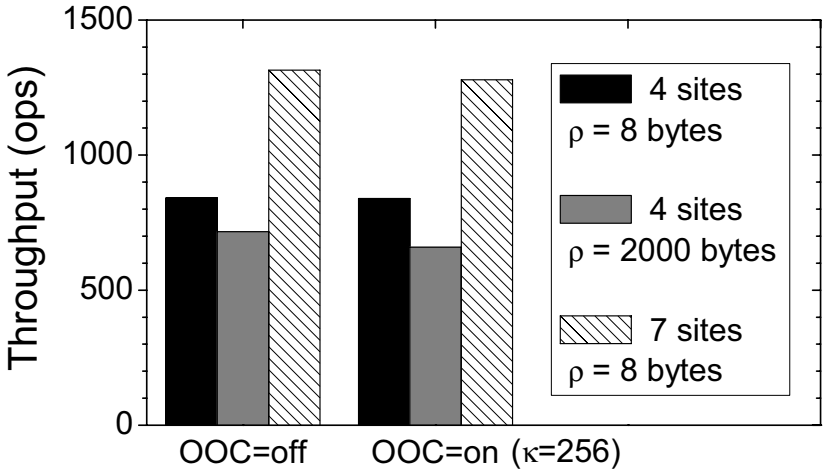


Figure 6.6: The throughput of RAM

A2M thread can sign approximate 220 messages per second. In the ideal case in which no skip messages are generated and signed at the any of the servers, the system should have maximal throughput of 880 ops. Indeed, RAM had a peak throughput of approximately 820 ops with four sites and $\rho = 8$ (black bars). Increasing ρ to 2,000 bytes did not show a significant throughput drop, which falls approximately 700 ops (gray bars).

We also ran experiments with seven sites, but due to hardware constraints, we used a star topology: each node had a 20 Mbps link to a central node with 25 ms one way delay. We only showed result for $\rho = 8$, since $\rho = 2000$ would have exceeded the 20 Mbps link bandwidth. The shaded bars showed 1320 ops throughput: a nearly linear scale from the four sites configuration ($1320/830 = 1.61 \approx 7/4 = 1.75$).

We have already seen in figure 6.5 that out-order-commit had little impact on throughput because the A2M is the bottleneck. Figure 6.6 confirms this result under different configurations.

Kotla *et al.* has reported 20 Kops throughput for PBFT and even higher throughput for Zyzzyva with comparable machine configuration in a LAN [33]. While RAM's throughput is significantly lower due to the more expensive signature operations, a lower throughput has been deemed sufficient in practical settings by others [15]. We see two

techniques that can be used to improve RAM's throughput: (1) use multi-threaded A2M implementation to take advantage of the availability of multi-core machines; and (2) batch small requests to amortize the cost of signatures at relatively low cost of increase in latency. For example, we project RAM to have $700 \times 10 = 7,000$ ops if a batch size of 10 is used for an average of 200 bytes client load. At the projected throughput, a client request would be buffered for at most $1000/(700/4) = 5.8$ ms before being proposed – which is small compared to the 100 ms round-trip latency, which is also the minimal latency for committing a request.

6.8.5 Revocation

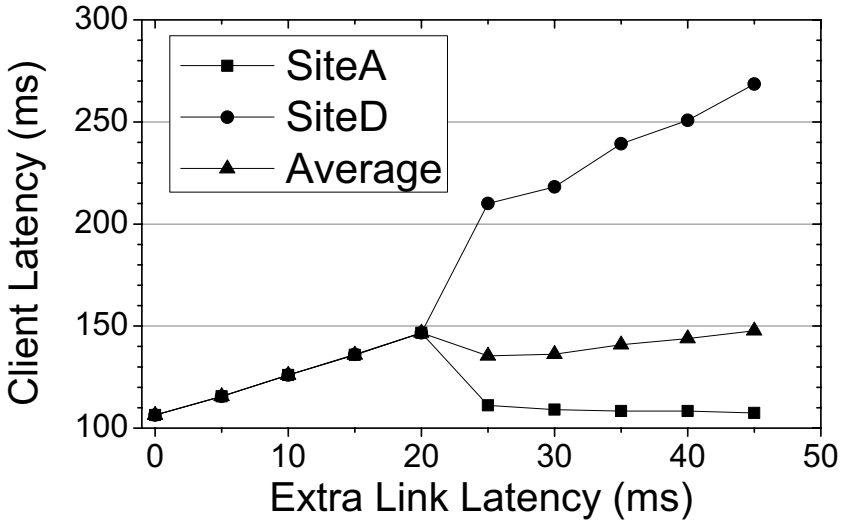


Figure 6.7: Increasing one site's latency leading to revocation.

Revocation has been used extensively in RAM. Understanding the implications of revocation is essential in selecting a good utility function U and a revocation policy P. In this section, we take a first step toward such a goal by implementing policy U_1 and P_1 and evaluating them under the following scenario. In a fully connected four-site setting (site A - D), we run one client at each site and configure each link to have 50

ms one-way delay. We then gradually add δ ms extra one-way delay to all links that connect to site D. When δ reaches 25 ms, D is considered too slow and hence revoked. While revoked, D uses A to forward its requests. Figure 6.7 shows the measured latency after the system stabilizes.

Before revocation occurred, the latency at all sites grew linearly with δ. During this period, if D had been revoked, sites A - C would have achieved a latency close to the lower bound. Revoking D, however, also increased the latency of D by an additional 50 ms. During this initial period, the amount of latency reduction that could be obtained by site A - C might not justify the latency increase at D. As δ continued to grow, D was revoked. Although D's latency grows considerably larger, the revocation is justified because: (1) D's latency was primarily due to the slow links incident on D, and (2) the other three sites were able to achieve the best possible latency and so the average latency of the system did not increase significantly as a result of the slow links to and from D. In fact, in this particular scenario, the average latency never grew over 150 ms even though the latency of D was almost doubled.

6.9 Related work

Fault-tolerant protocols. We have explained Paxos [37], Mencius, and PBFT [14]. FaB [47] reduces PBFT's proposing phase from two steps to one. The improved latency, however, comes at the cost of more replicas: $5f+1$ are required to tolerate f failures.

A 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. Speculation often requires the replicated state machine to tentatively executes a request and roll back its state if necessary. Client speculation has also being used to both improve latency [64]. Doing so though requires the clients being able to rollback their states and do not output any unstable result to the user. The way RAM handles faulty A2M is similar to such an approach. Finally, when properly used speculation can also be used to improve throughput. For example, *Zyzzyva* [33] improves system throughput by letting the clients handle tentatively executed result and decide if rollback is necessary. Doing so requires the clients being able to talk directly to all servers, which may not be

realistic for multi-site wide-area system due to security concerns.

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 [20].

Amir *et al.* [6] observe that a slow primary can reduce throughput or increase latency significantly. To deal with the problem, Amir *et al.* proposed a protocol *Prime* that uses a pre-agreement protocol that requires all-to-all communication and signatures. *Spin* is also a protocol that rotates the leader [62]. However, instead of pre-assigning instances (as is done with RAM), Spin uses a different leader upon every new batch of requests. One drawback of Prime and Spin is that they increase the latency observed by the clients even when there is no failure or false suspicion. RAM, on the other hand, does not sacrifices normal execution latency while still providing robust latency in the face of uncivil behavior. *Aardvark* [20] is a robust BFT protocol that provides usable throughput during both failure-free and uncivil runs. It accomplishes this goal by frequently changing the primary (using performance metrics), by signing client requests, and by isolating resources.

Steward [7] is the only BFT protocol we are aware that was 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. RAM is specifically designed to tolerate such scenarios by assuming remote processes are untrustworthy.

Other notable BFT protocols in the literature are *Q/U* [2] and *HQ* [23], both of which use quorum update protocols. Q/U uses a state-machine replication protocol based on quorum agreement that is more scalable than traditional state-machine replication protocols, but Q/U only not requires a higher degree of replication but also performs poorly in the presence of contention. HQ is a hybrid protocol that uses quorum agreement in the absence of contention, and uses PBFT to order contending operations. RAM already achieves low latency by design (clients receive a response in one WAN round-trip latency), and using quorum agreement to speed up the execution of operations is unnecessary.

RAM is also built to discourage uncivil rational behaviors. Similar philosophy

has been used to design applications under a different context [4].

Finally, RAM focuses on tolerating and detecting faulty servers. Tolerating faulty client behavior is an orthogonal problem and there has been solutions proposed in the literature to cope with Byzantine clients [23, 43].

Trusted components. The Wormholes model of Veríssimo [61] is a hybrid system model that defines a system comprising a subsystem with stronger guarantees (a privileged artifact) along with main functional component that provides weaker guarantees. Correia *et al.* discuss the design of the *TTCB* security kernel, which is a simple subsystem that provides a set of basic security services [22]. Neves *et al.* show how to use TTCB to solve vector consensus in an asynchronous system with Byzantine failures [50].

A2M [19] and TrInc [42] are trusted devices that prevent equivocation and enable more efficient protocols. With A2M, Chun *et al.* have shown that it is possible to implement a variant of PBFT that requires only $2f + 1$ replicas. RAM also uses A2M, but requires $3f + 1$ replicas. We opted for a higher degree of replication because it leads to a simpler and more efficient implementation of the protocol, and enables the detection of faulty A2M devices. TrInc is a simpler device that implements essentially a set of trusted counters and is shown can be used to implement A2M. The way TrInc and RAM use trusted hardware component shares the same philosophy: simplify the semantic to make implementing such trust devices more realistic.

Performance prediction. Performance prediction is an critical aspect of our approach as deviating from the expected performance may cause the revocation of a server. There has been work in the area of predicting performance of communication for wide-area systems. Schulz *et al.* propose an algorithm for predicting the performance of communication in wide-area parallel computing applications. *SyncProbe* provides applications with a real-time estimate of the maximum expected message delay [59]. Lu *et al.* with *DualPats* exploits the strong correlation between TCP throughput and flow size, and the statistical stability of Internet path characteristics to accurately predict the TCP throughput of large transfers using active probing [45]. On the performance of services, *Quorum* uses response monitoring among other mechanisms at the border of an Internet site to

ensure throughput and response time guarantees [12].

6.10 Future direction

We have explained the core mechanisms that enables RAM. The preliminary evaluation on our prototype implementation also demonstrates promising results. The following issues, however, need yet to be studied in the future.

Improving throughput: While we have demonstrated that RAM can provide realistic throughput and has the potential to improve by using multi-threading and/or batching, we need yet to fully investigate the effectiveness of such techniques. Having improved throughput makes RAM suitable for a wider range of applications.

Separating policy and mechanism: We have explain several rules for detecting uncivil rational and irrational behavior. These rules are, however, not mean to be complete. Different systems are also likely to have different rule for achieving their specific performance goals. So, additional rules need to be provided as policies so that RAM can be changed while deployed. Doing so needs to further sperate the mechanisms of RAM from policies both in protocol design and in implementation.

Utility functions and revocation policies: We have introduced example utility function and revocation policy to optimize system utility. A more comprehensive study is needed to further understand the potential, limitation, and tradeoff for choosing such functions and policies.

Checkpoint and recovery: RAM is designed to work well with checkpoint and recovery tools to protect against the tampering of A2Ms. We need yet to fully investigate the integration of RAM with such tools.

6.11 Summary

RAM is an efficient Byzantine fault-tolerant protocol for wide-area replication. There are three core concepts that RAM builds upon: a **R**otating leader design, the use of **A**2M devices to prevent equivocation, and a trust model based on **M**utually suspicious domains (MSD). Together, these concepts enable a protocol that provides low latency in wide-area systems. To prevent uncivil rational and irrational servers from disrupting the protocol execution for extended periods of time, RAM relies upon revocation, which removes the privilege of a server to directly propose new requests. Revocation, however, can be a double-edge sword, and can also harm civil servers if not used correctly. We have consequently crafted a mechanism based on performance estimates and suspicion certificates that strives to maximize system utilization. Our evaluation shows that RAM obtains excellent latency performance compared to PBFT, and realistic throughput. It also shows that servers are able to make appropriate decisions with respect to revocation.

6.12 Acknowledgement

This chapter is, in part, reprints of material as it appears in "Towards Low Latency State Machine Replication for Uncivil Wide-area Networks", by Yanhua Mao, Flavio Junqueira, and Keith Marzullo, in the Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep), June, 2009 and "RAM: State Machine Replication for Uncivil Wide-area Networks", by Yanhua Mao, Flavio Junqueira, and Keith Marzullo, unpublished technical report, Department of Computer Science and Engineering, University of California, San Diego, October, 2009. The dissertation author was the primary coauthor and co-investigator of the two papers.

Chapter 7

Conclusion

This dissertation studied protocols that implement state machine replication for wide area networks, specifically for multi-site wide area networks.

State machine replication is the most general approach for providing highly available stateful services. By replicating deterministic services across a few replicas, the service being replicated is available as long as the total number of failures does not exceed a certain threshold. If the replicas fail independently, then this approach can significantly increase the availability and reliability of the service by replicating less available and reliable servers.

State machine replication is also widely applicable because it provides strong consistency, which guarantees that the clients that use the replicated service always observe the same sequence of state change. Doing so greatly simplifies the design and implementation of clients: the replication layer is transparent to the core client logic, so, the clients do not have to deal with concurrency themselves.

The core building block of state machine replication is consensus, which, despite the possibility of contention and failures, solves the problem of how replicas agree on a unique command from those sent by clients. By running an unbounded sequence of instances of consensus, it is possible to select a consistent sequence of commands across all the correct replicas.

Like many other distributed computing problems, both consensus and replicated state machine are defined using a set of safety and liveness properties. The safety properties state the conditions that an implementation must satisfy at all time: they allow

one to verify that nothing bad has happened yet. The liveness properties, on the other hand, state the condition that an implementation must eventually satisfy: they require that something good eventually happens.

Consensus is a well-studied problem: many protocols have been proposed to solve consensus under different system models, failure models and/or assumptions. The asynchronous message passing model, in which there is no upper bound on either the message deliver delays or the message processing delays, is popular due to its generality: most modern operating systems and communication networks are modeled as asynchronous. It is, however, well-known that consensus cannot be both safe and live in an asynchronous system with the possibility of just a single failure. This is because of the inherent difficulty to distinguish a faulty replica from a slow replica in an asynchronous system.

There are many ways to circumvent this impossibility result. The most popular approach is to use unreliable failure detectors, which are classified in term of their accuracy and completeness properties. Despite the possibility of an unbounded number of mistakes made by the failure detector, one can design protocols that satisfy safety at all time and provide liveness when the failure detector becomes accurate. The leader electron failure detector class, denoted as Ω, has been shown to be the weakest failure detector for providing both safety and liveness for consensus. Many existing consensus protocols, such as Paxos and PBFT, are designed to work with Ω due to its simple semantics.

Despite the advantages of wide-area deployment, such as the ability to cope with co-located failures and to facilitate wide-area applications, replicated state machines are usually deployed in local area environments. Most existing protocols for consensus and replicated state machine are also designed with local area networks in mind, making them less efficient in the wide area. To make the replicated state machine approach more practical and applicable in the wide area, this dissertation focused on designing protocols that works both efficiently and effectively in such environments.

Following a brief review of background materials in chapter 2, this dissertation started the investigation with the crash failure model, in which the replicas may fail by stopping to execute.

To understand the challenges of designing efficient protocols, chapter 3 compared two crash failure protocols: Paxos and Fast Paxos. Paxos is a leader-based protocol that tolerates up to f failures with $2f+1$ replicas. It is popular due to its simplicity. Fast Paxos is a varient of Paxos that tolerates up to f failures with $3f+1$ replicas. It is a fast learning protocol that is designed to have lower latency in local area networks than Paxos. In the comparison, we considered a simple cluster-based wide-area system, in which all the servers are in the same local area network while the clients are in a remote network. Simply putting the clients in a remote network yields surprising analytic result: Paxos has a fixed probability of 60% of being faster in latency than Fast Paxos if (1) the local-area communication is negligible comparing to that of wide area and (2) the wide-area communication follows a distribution that is not concentrate on any value, *e.g.*, continuous. Our simulation results also confirmed that Paxos is faster in latency. This motivated us to examine the characteristics of the network architecture and to design new protocols based on them.

The remainder of the dissertation focused on multi-site wide-area systems. Such a system is consist of a small number of sites. Each site implements one server replica and multiple clients. Together, the servers implements a replicated state machine service. The clients access the service through their local servers. Chapter 4 presented $\mathscr{R}$, a generic rotating leader protocol for such multi-site systems. First, we introduce *simple consensus*, a variant of consensus that restricts the value a server can propose: a well-known server, called the *coordinator*, can propose any command while all other servers can only propose the special *no-op* command. We proved that simple consensus is equivalent to consensus in term of solvability: Ω is the weakest failure detector for solving both problems. We gave special names to the actions of a server can take: the coordinator *suggests* by proposing a command other than *no-op*; the coordinator *skips* by proposing the *no-op* command; and the followers (non-coordinators) *revokes* by proposing *no-op* on behave of the coordinator that is being suspected to have failed.

We then solved the replicated state machine problem by running an unbounded instances of simple consensus instead of consensus. We used a rotating-leader design. The idea is to partition the simple consensus sequence space so that each server is the coordinator for a sub-set of an unbounded number of the instances. For example, they

may be simply assigned in a round-robin manner.

Each instances of simple consensus is then implemented with $\mathscr{A}$, a coordinated simple consensus protocol that takes advantage of the more restricted definition of simple consensus. For example, $\mathscr{A}$ may assign the coordinator as the default leader of the instance. The details of protocol $\mathscr{A}$ depend on the failure model and system assumption. It has, though, one key characteristic – fast skipping, which allows all servers to immediately learn that *no-op* is the simple consensus outcome as soon as they confirm the coordinator has skipped. Fast skipping is a unique feature enabled by the more restricted definition of simple consensus.

With $\mathscr{A}$ implementing simple consensus, the generic protocol $\mathscr{R}$ works as follows: (1) Upon receiving commands from clients, a server immediately submit them to the next available simple consensus instances it coordinates; (2) A server skips its unused simple consensus instances when it observes other servers are consuming instances faster than itself; (3) A server revokes the coordinator when it suspect the coordinator has failed; (4) If the eventually perfect failure detector $\Diamond\mathbb{P}$ is available, to provide liveness, a server re-suggests outstanding requests when it recovers from a false suspicion or a failure; (5) If only the weaker Ω is available, to provide liveness, a server forwards outstanding requests to other servers when under a prolonged period of false suspicion.

The resulting generic protocol $\mathscr{R}$ is not efficient by itself. To improve its efficiency, we studied optimizations that are generic to the rotating leader design of $\mathscr{R}$: revocation in large block and out-of-order commit. Additional optimizations can also be applied to rotating leader protocols derived from $\mathscr{R}$, but usually depend on the specific protocol $\mathscr{A}$ that is used.

First, due to the expensive revocation process, $\mathscr{R}$ has both high message complexity and high latency when one or more servers have failed. To improve, we apply an optimization to allow revocation to be issued ahead of time and in large block size. Issuing revocation for future turns helps to improve the commit latency for non-faulty servers: learned commands can commit without having to wait for the revocation process to finish. Issuing revocation in large block size amortize the message complexity: the message exchanged for revocation becomes negligible in the long run.

The rotating leader design allows $\mathscr{R}$ to achieve minimal commit latency at all

servers when there are no concurrent requests. With concurrent requests, the delayed commit problem can occur: to commit, a learned command in a simple consensus instance has to wait until all prior instances have been learned and committed. The extra delay caused by delayed commit does not grow indefinitely: we prove an upper bound based on the latency metrics of $\mathscr{A}$. Such delay can also be mitigated using *out-of order commit*: a flexible commit mechanism that allows commutable requests to commit in any orders. Out-order-commit is another benefit of using simple consensus: restricting the value the followers can propose makes it easier to allow such flexible commit mechanism.

In chapter 5, we change our focus to designing an efficient protocol for crash-failure multi-site systems. We make little assumptions about the network characteristics or the system load for such a system. Indeed, we only assume that wide-area communications have higher latency and lower bandwidth than that of local area; system load may change from time to time and/or site to site; and wide-area bandwidth may vary over time and may not be symmetric. Our goal is to design a protocol that adapts well in such an environment and can deliver both high throughput under high load and low latency under low load.

The first step of our study is to compare existing protocols, such as Paxos, Fast Paxos and CoReFP, in multi-site systems. Not surprisingly, none of the protocols works perfectly. But Paxos has more advantages than others. This motivates us to derive our own protocol from Paoxs.

The idea is to apply Paxos to the generic rotating leader protocol $\mathscr{R}$. First, we derive Coordinated Paxos from Paxos to efficiently implement simple consensus. We then instantiate $\mathscr{R}$ using Coordinated Paxos. To improve efficiency, besides the general optimizations for $\mathscr{R}$, additional optimizations are applied to opportunistically piggybacking messages for skipping to other messages. Doing so makes it possible for the servers to skip at effectively no extra cost. The resulting protocol is called *Mencius*.

We evaluate the performance of Mencius by comparing it to that of Paxos. In a basic three-site configuration, the throughput of Mencius is 200% higher than that of Paxos under a network-bound load and 50% higher than that of Paxos under a CPU-bound load. Mencius also have better scalability than Paxos. With seven sites, the

throughput of Mencius is seven times that of Paxos under a network-bound load and two times that of Paxos under a CPU-bound load. We also showed that Mencius is able to automatically adapt to changing network bandwidth and use all available bandwidth for provide throughput. Mencius has lower latency than Paxos when there is little or no contention. Under high contention, the latency of Mencius is no worse than that of Paxos. Enabling out-of-order commit helps Mencius reduce latency by up to 30% when under high contention. The only case in which Mencius performances worse than Paxos is when a failure is not detected: Mencius temporarily loses liveness when *any* server has crashed but not yet been detected; Paxos temporarily loses liveness only when the leader has crashed but not yet been detected.

In chapter 6, we continue studying multi-site systems but under the more general Byzantine failure model, in which a server may fail in an arbitrary manner.

Performance wise, we seeked a design that provides low latency for all sites and realistic throughput. First, we compared PBFT, the de facto standard protocol for the Byzantine failure model, with Paxos and identify three main causes of the higher latency of PBFT. Next, we proposed three techniques for reducing latency for each of three causes. First, the rotating-leader design of $\mathscr{R}$ can be used to reduce the extra latency at the non-leader sites. Second, the Attested Append-only Memory (A2M), a specialized hardware, can be used to reduce the proposing phase of PBFT from two steps to just one step. Finally, the Mutually Suspicious Domains (MSD) model, which allows clients to trust their local servers, can be used to address the extra latency caused by the traditional flat Byzantine model, in which there is no trust between any pair of processes.

These three techniques make up the core building block of our protocol *RAM*. First we used A2M and MSD to implement Coordinated Byzantine Paxos, which in turn implements simple consensus. We then used Coordinated Byzantine Paxos to instantiate the generic rotating leader protocol $\mathscr{R}$. Finally, we obtained the protocol RAM By applying optimizations to improve efficiency.

Assuming the Byzantine failure model not only allows the applications to tolerate a wider range of failures but also shift our design goals. Instead of only providing good performance with the benign crash failure, we need to design protocols that can

cope with potential uncivil and/or malicious behavior. We roughly classified the behavior of a server into three categories: *civil*, *uncivil rational* and *irrational*. A civil behavior is consider as correct while both uncivil rational and irrational behavior are faulty. Uncivil rational and irrational behavior have different root causes: uncivil rational behavior is caused by selfish/faulty administrators that try to improve local utility (in our case, the latency of local requests) by gaming the system; irrational behavior can arise due to remote attack that tries to disturb the system. Differences in root causes result in different strategies for cope with such behavior.

For uncivil rational behavior, we discouraged it by designing the protocol such that any divergence from the correct behavior will only increase the chance of lowering local utility. This way, the best strategy for a rational server is to follow the protocol. We did this with revocation, which penalizes uncivil behavior by removing a server's ability to propose commands directly. Doing this increases the server's latency. We demonstrated how to use revocation to discourage a wide range of uncivil rational behavior, such as untimely behavior, forwarding omission, flooding and irresponsible suspicion.

For irrational behavior, we designed RAM to identify as much faulty behavior as possible. We demonstrated how irrational behavior such as compromised A2M, faulty local server and ignoring revocation can be detected. When detecting is impossible, we resort to damage control method that bound the latency increase that can be caused by undetected Byzantine servers. By providing example utility estimation function and revocation policy, we proved upper bounds for request commit delays.

Finally we evaluated the performance of RAM with our prototype implementation. Our results showed that RAM has a latency by up to 50% lower than that of PBFT. RAM does, though, have lower throughput than PBFT due to its use of digital signatures for failure-detecting purpose. The throughput of RAM is, however, still higher than that have claimed as being sufficient for practical applications. Adding additional replicas also helps to improve the system throughput. Finally, we showed the use of a utility estimation function and corresponding policy in preserving high system utility (low latency) in the face of a server that becomes unusually slow.

There are several future directions of research, primarily in the context of RAM. These include:

- Improving throughput performance through batching and multi-core A2M implementation.
- Integrating RAM with a checkpoint and recovery mechanism to cope with faulty A2M and irrational local servers.
- Separating the policy RAM uses for utility and for failure detection from the mechanisms of RAM. This would allow RAM to be dynamically tunable based on a changing environment.
- Running RAM in collaboration with a broader security environment, such as a firewall rule manager or a network intrusion detection system.

Bibliography

[1] *Secure Hash Standard*. National Institute of Standards and Technology, Washington, 2002. Federal Information Processing Standard 180-2.

[2] 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.

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

[4] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, volume 39, pages 45–58, New York, NY, USA, December 2005. ACM Press.

[5] 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.

[6] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Byzantine replication under attack. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 197–206, Anchorage, AK, USA, 2008. IEEE Computer Society.

[7] Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Scaling Byzantine fault-tolerant replication to wide area networks. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 105–114, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[9] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–15, London, UK, 1996. Springer-Verlag.

[10] Terry Benzel, Robert Braden, Dongho Kim, Clifford Neuman, Anthony Joseph, Keith Sklower, Ron Ostrenga, and Stephen Schwab. Design, deployment, and use of the deter testbed. In *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*, pages 1–1, Berkeley, CA, USA, 2007. USENIX Association.

[11] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-crummey, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS project: Software support for high-level Grid application development. *International Journal of High Performance Computing Applications*, 15:327–344, 2001.

[12] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum: flexible quality of service for internet services. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 159–174, Berkeley, CA, USA, 2005. USENIX Association.

[13] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[14] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[15] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[16] Tushar Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

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

[18] Francis C. Chu. Reducing Ω to $\Diamond W$. *Information Processing Letters*, 67(6):289–293, 1998.

[19] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 189–204, New York, NY, USA, 2007. ACM.

[20] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In

Proceedings of the 6th USENIX symposium on Networked systems design and implementation, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[21] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[22] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proc. of the Fourth European Dependable Computing Conference*, October 2002.

[23] 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.

[24] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Survey*, 36(4):372–421, 2004.

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

[26] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–7, New York, NY, USA, 1983. ACM.

[27] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[28] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quema. High throughput total order broadcast for cluster environments. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 549–557, Washington, DC, USA, 2006. IEEE Computer Society.

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

[30] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transcation on Programming Language System*, 12(3):463–492, 1990.

[31] Flavio Junqueira and Keith Marzullo. Coterie availability in sites. In *In DISC*, pages 3–17. Springer Verlag, 2005.

[32] Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 169–178, New York, NY, USA, 2006. ACM Press.

[33] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *SIGOPS Operting System Review*, 41(6):45–58, 2007.

[34] 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.

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

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

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

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

[39] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[40] Leslie Lamport, Aamer Hydrie, and Demetrios Achlioptas. Multi-leader distributed system. U.S. patent 7,260,611 B2, Aug 2007.

[41] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[42] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: small trusted hardware for large distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.

[43] Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *26th IEEE International Conference on Distributed Computing Systems*, pages 34–43, Lisbon, Portugal, Jul 2006.

[44] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. *SIGOPS Oper. Syst. Rev.*, 40(4):103–115, 2006.

[45] Dong Lu, Yi Qiao, Peter A. Dinda, and Fabian E. Bustamante. Characterizing and predicting TCP throughput on the wide area network. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 414–424, Washington, DC, USA, 2005. IEEE Computer Society.

[46] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[47] Jean-Philippe Martin. Fast Byzantine consensus. *IEEE Transcation on Dependable Security Compututing*, 3(3):202–215, 2006. Senior Member-Alvisi, Lorenzo.

[48] Rich Miller. Explosion at the planet causes major outage. `http://www.datacenterknowledge.com/archives/2008/06/01/explosion-at-the-planet-causes-major-outage/`. June 1, 2008 (accessed Febuarary 15, 2010).

[49] John Nagle. RFC 896: Congestion control in IP/TCP internetworks, January 1984. Status: UNKNOWN.

[50] Nuno F. Neves, Miguel Correia, and Paulo Veríssimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

[51] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Analysis of computer intrusions using sequences of function calls. *IEEE Transactions on Dependable and Secure Computing*, 4(2):137–150, 2007.

[52] R. Rivest. RFC1321: The MD5 message-digest algorithm, April 1992.

[53] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

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

[55] Martin Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[56] Livia Sampaio and Francisco Brasileiro. Adaptive indulgent consensus. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 422–431, Washington, DC, USA, 2005. IEEE Computer Society.

[57] Rodrigo Schmidt, Lásaro Camargos, and Fernando Pedone. On collision-fast atomic broadcast. Technical Report LABOS-REPORT-2007-001, Éole Polytechnique Fédérale de Lausanne, 2007.

[58] Fred B. Schneider and Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22:299–319, 1990.

[59] Jawwad Shamsi and Monica Brockmeyer. SyncProbe: Providing assurance of message latency through predictive monitoring of Internet paths. *High-Assurance Systems Engineering, IEEE International Symposium on*, 0:187–196, 2007.

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

[61] Paulo E. Veríssimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[62] Giuliana Veronese, Miguel Correia, Alysson Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *SRDS'09: The 30th IEEE Symposium on Reliable Distributed Systems*, Niagara Falls, USA, September 2009.

[63] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Milliar-Smith, R. E. Shostak, and C. B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. pages 560–575, 1989.

[64] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 245–260, Berkeley, CA, USA, 2009. USENIX Association.

[65] Rich Wolski. Experiences with predicting resource performance on-line in computational grid settings. *SIGMETRICS Perform. Eval. Rev.*, 30(4):41–49, 2003.

[66] Jingmin Zhou, Mark Heckman, Brennen Reynolds, Adam Carlson, and Matt Bishop. Modeling network intrusion detection alerts for correlation. *ACM Trans. Inf. Syst. Secur.*, 10(1):4, 2007.

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

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

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

Appendix A

Pseudo code of Coordinated Paxos

/* Pseudo code of Coordinated Paxos at server p.
Each round of Coordinated Paxos is assigned to one of the servers. The round number is also called ballot number.
Function $owner(r)$ returns the server ID of the owner of round (ballot number) r.
Note that this is only the pseudo code for one instance of Coordinated Paxos.
PREPARE(b): PREPARE message for ballot number b.
ACK(b, ab, av): to acknowledge **PREPARE**(b): ab is the highest ballot the sending server has accepted a value, and av is the value accepted for ballot ab.
PROPOSE(b, v): to propose a value v with ballot number b.
ACCEPT(b, v): to acknowledge **PROPOSE**(b, v) that the sending server has accepted v for ballot number b.
LEARN(v): to inform other servers that value v has been chosen for this instance of simple consensus. */

/* learner states: */

variable $learned \leftarrow \perp$; // No value is learned initially.

variable $learner\_history \leftarrow \{\}$ // No peer has accepted any value.

/* proposer states: */

variable $prepared\_history \leftarrow \{\}$ // No prepared history initially.

/* acceptor states: */

variable $prepared\_ballot \leftarrow 0$ // All servers are initially prepared for ballot number 0.

variable $accepted\_ballot \leftarrow -1$ // Initially, no ballot is accepted.

variable $accepted\_value \leftarrow \perp$ // Initially, no value is accepted.

/* Coordinator can call either SUGGEST or SKIP. */

procedure SUGGEST(v) // Coordinator proposes value v.

Broadcast PROPOSE$(0, v)$

end procedure

procedure SKIP // Coordinator proposes *no-op*.

Broadcast PROPOSE$(0, no\text{-}op)$

end procedure

procedure REVOKE // Non-coordinator starts to propose *no-op*.

$ballot \leftarrow$ **Choose** $b : owner(b) = p \wedge b > prepared\_ballot \wedge b > accepted\_ballot$

/* Choose a ballot that is owned by p and greater than other ballot p has ever seen. */

Broadcast PREPARE$(ballot)$ // Start phase 1 with a higher ballot number.

end procedure

OnMessage ANY From q **OnCondition** $learned \neq \perp$

Begin

if *the incoming message is not a* LEARN *message* **then**

Send LEARN$(learned)$ **To** q

end if

End

OnMessage PREPARE(b) **From** q **OnCondition** $learned = \perp$

Begin

if $b > prepared\_ballot$ **then**

$prepared\_ballot \leftarrow b$

Send ACK$(b, accepted\_ballot, accepted\_value)$ **To** q

end if

End

OnMessage LEARN(v) **From** q **OnCondition** $learned = \perp$

Begin

$learned \leftarrow v$

ONLEARNED(v)

End

OnMessage ACCEPT(b,v) **From** q **OnCondition** $learned = \perp$

Begin

if $b = 0$ **then**

/* This ACCEPT message acknowledges a SUGGEST message. */

ONACCEPTSUGGESTION(q) // An upcall interface for Mencius.

end if

$learner\_history \leftarrow learner\_history \cup \{\langle b,v,q\rangle\}$

$LSet \leftarrow \{\langle e_1,e_2,e_3,\rangle : e_1 = b \wedge \langle e_1,e_2,e_3\rangle \in learner\_history\}$

if $size(LSet) = \lceil (n+1)/2 \rceil$ **then**

/* a quorum has accepted the value, the value is chosen. */

Broadcast LEARN(v)

end if

End

OnMessage ACK(b,a,v) **From** q **OnCondition** $learned = \perp$

Begin

$prepared\_history \leftarrow prepared\_history \cup \{\langle b,a,v,q\rangle\}$

$PSet \leftarrow \{\langle e_1,e_2,e_3,e_4\rangle : e_1 = b \wedge \langle e_1,e_2,e_3,e_4\rangle \in prepared\_history\}$

if $size(PSet) = \lceil (n+1)/2 \rceil$ **then**

/* A quorum of peers have been prepared, ready to propose. */

$ha \leftarrow max\{a : \langle -,a,-,-\rangle \in PSet\}$

$hvset \leftarrow \{v : \langle -,ha,v,-\rangle \in PSet\}$

$hv \leftarrow$ **Choose** $v : v \in hvset$

/* hv is set to the only element in $hvset$, since $hvset$ must have one unique element. */

if $hv = \perp$ **then**

/* No value has been chosen yet, propose *no-op*. */

Broadcast PROPOSE$(b, no\text{-}op)$

else

/* Must propose hv. */

BroadcastPROPOSE(b, hv)

end if

end if

End

OnMessage PROPOSE(b, v) **From** q **OnCondition** $learned = \perp$

Begin

if $b = 0 \wedge v = no\text{-}op$ **then**

/* Coordinator skips, p learns *no-op* immediately */

$learned \leftarrow no\text{-}op$

ONLEARNED(*no-op*)

else if $prepared\_ballot \leq b \wedge accepted\_ballot < b$ **then** // p accepts (b, v).

if $b = 0$ **then**

/* this is a SUGGEST message. */

ONSUGGESTION // An upcall interface for protocol $\mathscr{P}$ and Mencius.

end if

$accepted\_ballot \leftarrow b$

$accepted\_value \leftarrow v$

Send ACCEPT(b, v) **To** q

end if

End

Appendix B

Pseudo code of Protocol $\mathscr{P}$

/* Pseudo code of Protocol $\mathscr{P}$ at server p.
Protocol $\mathscr{P}$ runs a series of Coordinated Paxos. It commits a value learned in instance i once all instances prior to i have been learned and committed. The following code handles duplication by checking for duplications before committing. Other techniques, such as assuming idempotent requests, can also be used.
Note that besides the original arguments, all upcalls/downcalls from/to Coordinated Paxos also have an additional argument i that specifies the simple consensus instance number.
Protocol $\mathscr{P}$ provides two APIs to its applications. The applications downcalls ONCLIENTREQUEST to submit a request to the state machine. When a value is chosen, $\mathscr{P}$ upcalls ONCOMMIT to notify the application.
Function $owner(i)$ returns the coordinator of instance i.
Function $learned(i)$ returns a reference to the $learned$ variable of the i^{th} simple consensus instance. */

variable $proposed[\,]$ // An array records the value that the coordinator initially suggested to an instance. It maps an instance number to a value. Every key is initially mapped to $\perp$.

variable $index \leftarrow min\{i : owner(i) = p\}$ // The next instance to suggest a value to.

variable $expected \leftarrow 0$ // The next instance number to commit a value, *i.e.*, the smallest instance whose value is not learned.

procedure ONCLIENTREQUEST(v) SUGGEST($index, v$) // Rule 1: p suggests v to instance $index$.

$proposed[index] \leftarrow v$

$index \leftarrow min\{i : owner(i) = p \wedge i > index\}$

end procedure

/* Rule 2: on receiving a SUGGEST message for instance i, p skips all unused instances prior to i. */

procedure ONSUGGESTION(i)

$SkipSet \leftarrow \{k : k \geq index \wedge k < i \wedge owner(k) = p\}$

for all k **in** $SkipSet$ **do**

SKIP(k) // Skip instance k

$index \leftarrow min\{k : owner(k) = p \wedge k > i\}$

end for

end procedure

/* Rule 3: When suspecting q has failed, p revoke all instances that are smaller than $index$ and are coordinated by q. */

procedure ONSUSPECT(q)

$RevokeSet \leftarrow \{i : owner(i) = q \wedge i < index \wedge learned(i) = \perp\}$

for all k **in** $RevokeSet$ **do**

REVOKE(k) // Revoke instance k.

end for

end procedure

procedure CHECKCOMMIT // Check if a new value can be committed.

while $learned(expected) \neq \perp$ **do**

$v \leftarrow learned(expected)$

if $v \neq$ no-op $\wedge\, v \notin \{learned(i) : 0 \leq i < expected\}$ **then**

/* Commit value v only if it is not a *no-op* and is not a duplication. */

ONCOMMIT(v)

end if

$expected \leftarrow expected + 1$

```
    end while
end procedure

procedure ONLEARNED(i, v)          // Upon instance i learns value v.
    if owner(i) = p ∧ proposed[i] ≠ v then
        /* Rule 4: v must be no-op and proposed[i] must be re-suggested. */
        ONCLIENTREQUEST(proposed[i])
    end if
    CHECKCOMMIT
end procedure
```

Appendix C

Pseudo code of Mencius

/* Pseudo code of Mencius at server p. Mencius runs a series of Coordinated Paxos. It commits a value learned in instance i once all instances smaller than i have been learned and committed. The following code handles duplication by checking for duplications before committing. Other techniques, such as assuming idempotent requests, can also be used.
Mencius provides two APIs to its applications. The applications downcalls ONCLIENTREQUEST to submit a request to the state machine. When a value is chosen, Mencius upcalls ONCOMMIT to notify the application.
Note that besides the original arguments, all upcalls/downcalls from/to Coordinated Paxos also have an additional argument i that specifies the simple consensus instance number.
Function $owner(i)$ returns the coordinator of instance i.
Function $learned(i)$ returns a reference to the $learned$ variable of the i^{th} simple consensus instance. Mencius also uses n timers for Accelerator 1. */

variable $proposed[\,]$ // An array records the value that the coordinator initially suggested to an instance. It maps an instance number to a value. Every key is initially mapped to $\perp$.

variable $expected \leftarrow 0$ // The next instance number to commit a value, *i.e.*, the smallest instance whose value is not learned.

variable $index \leftarrow min\{i : owner(i) = p\}$ // The next instance to suggest a value to.

variable $est\_index[\,]$ // An array records the estimated index of other servers. It maps a server ID to an instance number. Initially, $est\_index[q] \leftarrow min\{i : owner(i) = q\}$.

variable $need\_to\_skip[\,]$ // An array records the set of outstanding SKIP messages

need to be sent to a server. It maps a server ID to a set of instance numbers. Every key is initially mapped to an empty set.

procedure ONCLIENTREQUEST(v)

/* By Optimization 2, SKIP messages will be piggybacked on SUGGEST messages. So, we cancel the timers that were previously set for Accelerator 1 and reset the records of the outstanding SKIP messages. */

for all q **in** $\{0,\ldots,n-1\}$ **do**

CANCELTIMER(q) // Cancel the q^{th} timer.

$need\_to\_skip[q] \leftarrow \{\}$

end for

SUGGEST($index, v$) // Rule 1: p suggests v to instance $index$.

$proposed[index] \leftarrow v$

$index \leftarrow min\{i : owner(i) = p \wedge i > index\}$

end procedure

procedure ONACCEPTSUGGESTION(i, q) // Upon receiving an ACCEPT message that acknowledges a previous SUGGEST message.

$QSkipSet \leftarrow \{j : est\_index[q] \leq j < i \wedge owner(j) = q\}$ // By Optimization 1: SKIP messages are piggybacked on this ACCEPT message. $QSkipSet$ is the set of instances q has skipped.

for all j **in** $QSkipSet$ **do**

$learned(j) \leftarrow no\text{-}op$

CHECKCOMMIT

end for

$est\_index[q] \leftarrow min\{j : j > i \wedge owner(j) = q\}$

end procedure

procedure ONSUGGESTION(i)

/* Upon receiving a SUGGEST message for instance i. */

$q \leftarrow owner(i)$ // q is the sender of the SUGGEST message.

$QSkipSet \leftarrow \{j : est\_index[q] \leq j < i \wedge owner(j) = q\}$ // By Optimization 2, SKIP messages are piggybacked on this SUGGEST message. $QSkipSet$ is the set of instances q has skipped.

for all j **in** $QSkipSet$ **do**

$learned(j) \leftarrow no\text{-}op$

end for

CHECKCOMMIT

$est\_index[q] \leftarrow min\{j : j > i \wedge owner(j) = q\}$

$SkipSet \leftarrow \{j : j \geq index \wedge j < i \wedge owner(j) = p\}$ // By Rule 2, server p skips all unused instances smaller than i. $SkipSet$ is the set of instances p needs to skip.

for all k **in** $SkipSet$ **do** $learned[k] \leftarrow no\text{-}op$

end for

CHECKCOMMIT

// p does not send SKIP messages to other servers immediately. Optimization 1: p piggyback the SKIP message to q on the ACCEPT message.

for all k **in** $\{r : 0 \leq r < n-1 \wedge r \neq p \wedge r \neq q\}$ **do** // Optimization 2: For all other servers, SKIP messages are not sent immediately, instead they wait for a future SUGGEST message.

if $need\_to\_skip[k] = \{\}$ **then** // Set timer for Accelerator 1.

$need\_to\_skip[k] \leftarrow SkipSet$

SETTIMER$((k, \tau))$ // Set the k^{th} timer to trigger at τ unit time from now.

else

$need\_to\_skip[k] \leftarrow need\_to\_skip[k] \cup SkipSet$

end if // Check if the number of outstanding SKIP is greater than α.

if $size(need\_to\_skip[k]) > \alpha$ **then** // By Accelerator 1, need to propagate the SKIP messages when the outstanding SKIP messages is larger than α.

SENDSKIP(k) // propagate the SKIP messages to server k.

end if

end for

$index \leftarrow min\{j : owner(j) = p \wedge j > i\}$

end procedure

procedure ONSUSPECT(q)

/* Rule 3 and Optimization 3: p revokes q for large block of instances, when suspecting server q has failed. */

$C_q = min\{i : owner(i) = q \wedge learned(i) = \perp\}$

if $C_q < index + \beta$ **then**

$RevokeSet \leftarrow \{i : C_q \leq i \leq index + 2\beta \wedge owner(i) = q \wedge learned(i) = \perp\}$

for all k **in** $RevokeSet$ **do**

REVOKE(k) // Revoke instance k.

end for

end if

end procedure

procedure ONLEARNED(i, v)

/* Upon instance i learns value v. */

if $owner(i) = p \wedge proposed[i] \neq v$ **then**

/* Rule 4: v must be *no-op* and $proposed[i]$ must be re-suggested. */

ONCLIENTREQUEST($proposed[i]$)

end if

CHECKCOMMIT;

end procedure

procedure ONTIMEOUT(k) // The k^{th} timer times out.

SENDSKIP(k) // propagate the SKIP messages to server k.

end procedure

procedure SENDSKIP(k)

CANCELTIMER(k) // Cancel the k^{th} timer.

for all $q \in need\_to\_skip[k]$ **in**

doSKIP(q)

end for

$need\_to\_skip[k] \leftarrow \{\}$

end procedure

procedure CHECKCOMMIT // Check if a new value can be committed.

while $learned(expected) \neq \perp$ **do**

$v \leftarrow learned(expected)$

if $v \neq$ no-op $\wedge\, v \notin \{learned(i) : 0 \leq i < expected\}$ **then** // Commit value v only if it is not a *no-op* and is not a duplication.

ONCOMMIT(v)

end if

$expected \leftarrow expected + 1$

end while

end procedure

Appendix D

Pseudo code of Coordinated Byzantine Paxos

/* Pseudo code of Coordinated Byzantine Paxos at server p.
Note that this is only the pseudo code for one instance of Coordinated Byzantine Paxos.
Each round of Coordinated Byzantine Paxos is assigned to one of the servers.
Function $owner(r)$ returns the server ID of the owner of round r.
$\langle m \rangle_{\alpha_p}$: a message m from p authenticated using the shared secret keys between p and other processes.
$\langle m \rangle_{\sigma_p}$: a message m signed by server p.
$\langle m \rangle_{\rho_p}$: a message m signed by the A2M of server p.
$\langle \text{SUGGEST}, v \rangle_{\rho_p}$: the coordinator p suggests v, *i.e.*, proposing $v \neq$ *no-op* to round 0.
$\langle \text{SKIP} \rangle_{\rho_p}$: the coordinator p skips, *i.e.*, proposing *no-op* to round 0.
$\langle \text{NEW-LEADER}, rc, r \rangle_{\alpha_p}$: a new leader of round r is elected to revoke the current leader. rc is the revocation certificate.
$\langle \text{ACCEPT}, v, r \rangle_{\alpha_p}$: a value v is accepted in round r by server p.
$\langle \text{ACK}, r', v, s, r \rangle_{\sigma_p}$: p has accepted value v in round r. s is a set of signatures that verifies v. p also promises not to accept any value in a round smaller than r' in the future.
$\langle \text{PRE-PREPARE}, pc, v, r \rangle_{\sigma_p}$: the new leader p informs other servers the proposal v for round r. pc is the progress certificate that vouches for v.
$\langle \text{PREPARE}, v, r \rangle_{\sigma_p}$: server p has prepared v in round r. */

/* learner states: */

variable $learned \leftarrow \perp$ // No value is learned initially.

variable $history_{learner} \leftarrow \{\}$ // Not peer has accepted any value

/* proposer states: */

variable $history_{prepare} \leftarrow \{\}$ // No prepare history

variable $round_{revoking} \leftarrow -1$ // Current revocation round

/* acceptor states: */

variable $round \leftarrow 0$ // current round, initially 0: No value will be accepted for a round smaller than the current round.

variable $accepted_{value} \leftarrow \perp$ // Initially, no value is accepted.

variable $accepted_{round} \leftarrow -1$ // The round in which a value, if any, is accepted.

variable $accepted_{signature} \leftarrow \{\}$ // A set of signatures that verifies the accepted value

variable $preprepared \leftarrow -1$ // The highest round in which a value is prepared

procedure SUGGEST(v) // The coordinator proposes value v

Broadcast $\langle \text{SUGGEST}, v \rangle_{\rho_p}$ // SUGGEST messages are relayed.

end procedure

procedure SKIP // The coordinator proposes *no-op*.

Broadcast $\langle \text{SKIP} \rangle_{\rho_p}$ // SKIP messages are relayed.

end procedure

procedure REVOKE(rc) // Non-coordinator starts to revoke once a revocation certificate rc is acquired.

/* Choose a round that is owned by p and is greater than any round p has ever seen. */

$r \leftarrow$ **Choose** $b : onwer(b) = p \wedge b > round \wedge b > accepted_{round}$

$round_{revoking} \leftarrow r$

Broadcast $\langle \text{NEW-LEADER}, rc, r \rangle_{\alpha_p}$

end procedure

OnMessage $\langle \text{SKIP} \rangle_{\rho_q}$ **From** q **OnCondition** q is the coordinator

Begin

$learned \leftarrow$ *no-op*

ONLEARNED(*no-op*)

End

OnMessage $\langle \text{SUGGEST}, v \rangle_{\rho_q}$ **From** q **OnCondition** q is the coordinator

Begin

if $accepted_{round} < 0$ **then**

$accepted_{value} \leftarrow v$

$accepted_{round} \leftarrow 0$

$accepted_{signature} \leftarrow \{\rho_q\}$

Broadcast $\langle \text{ACCEPT}, v, 0 \rangle_{\alpha_p}$

end if

End

OnMessage $\langle \text{ACCEPT}, v, r \rangle_{\alpha_q}$ **From** q

Begin

$history_{learner} \leftarrow history_{learner} \cup \{r, v, q\}$

$LSet \leftarrow \{\langle e_1, e_2, e_3 \rangle : e_1 = r \wedge \langle e_1, e_2, e_3 \rangle \in history_{learner}\}$

/* A value is learned if $2f+1$ matching accept messages are collected. */

if $size(LSet) = 2f + 1$ **then**

$learned \leftarrow v$

ONLEARNED(v)

end if

End

OnMessage $\langle \text{NEW-LEADER}, rc, r \rangle_{\alpha_q}$ **From** q

Begin

if $r > round$ **then**

$round \leftarrow r$

Send $\langle \text{ACK}, r, accepted_{value}, accepted_{signature}, accepted_{round} \rangle_{\sigma_p}$ **To** q

end if

End

OnMessage $\langle \text{ACK}, r, v, s, vr \rangle_{\alpha_q}$ **From** q
Begin
  if $r = round_{revoking}$ **then**
    $history_{ack} \leftarrow history_{ack} \cup \{\langle r, v, s, vr \rangle\}$
    $pc \leftarrow \{\langle e_1, e_2, e_3, e_4 \rangle : e_1 = r \wedge \langle e_1, e_2, e_3, e_4 \rangle \in history_{ack}\}$
    /* $2f+1$ ACK messages are needs to construct a progress certificate */
    if $Size(pc) = 2f + 1$ **then**
      $pc' \leftarrow \{\langle e_1, e_2, e_3, e_4 \rangle : e_2 \neq \perp \wedge \langle e_1, e_2, e_3, e_4 \rangle \in pc\}$
      /* if no value has been accepted in pc, the new proposal is *no-op*. */
      $vv \leftarrow no\text{-}op$
      if $pc' \neq \phi$ **then**
        /* otherwise choose the value with the highest round number */
        $m \leftarrow max\{vr : \langle -, -, -, vr \rangle \in pc'\}$
        $vv \leftarrow$ **Choose** $\{e_2 : e_3 = m \wedge \langle e_1, e_2, e_3, e_4 \rangle \in pc'\}$
      end if
      Broadcast $\langle \text{PRE-PREPARE}, pc, vv, r \rangle_{\sigma_p}$
    end if
  end if
End

OnMessage $\langle \text{PRE-PREPARE}, pc, v, r \rangle_{\sigma_q}$ **From** q
Begin
  /* A server prepares for at most one value in each round */
  if $preprepared < r \wedge \text{IsValid}(pc, v, r)$ **then**
    $round \leftarrow r$
    $preprepared \leftarrow r$
    Broadcast $\langle \text{PREPARE}, v, r \rangle_{\sigma_p}$
  end if
End

OnMessage $\langle \text{PREPARE}, v, r \rangle_{\sigma_q}$ **From** q

Begin

$history_{prepare} \leftarrow history_{prepare} \cup \{\langle v, r, q, \sigma_q \rangle\}$

$PSet \leftarrow \{\langle e_1, e_2, e_3, e_4 \rangle : e_1 = r \wedge \langle e_1, e_2, e_3, e_4 \rangle \in history_{prepare}\}$

/* A value is accepted if $2f+1$ matching PREPARE messages are collected. */

if $Size(PSet) = 2f + 1$ **then**

Broadcast $\langle \text{ACCEPT}, v, r \rangle_{\alpha_p}$

$accepted_{value} \leftarrow v$

$accepted_{round} \leftarrow r$

$accepted_{signature} \leftarrow \{\sigma : \langle -, -, -, \sigma \rangle \in PSet\}$

end if

End