Efficient and Scalable Replication of Services over Wide-Area Networks

Thesis by
Abdallah Abouzamazem

In Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy

University of Newcastle upon Tyne
Newcastle upon Tyne, UK
Submitted 9th March 2012

Abstract

Service replication ensures reliability and availability, but accomplishing it requires solving the total-order problem of guaranteeing that all replicas receive service requests in the same order. The problem, however, cannot be solved for a specific combination of three factors, namely, when (i) the message transmission delays cannot be reliably bounded, as often the case over wide-area networks such as the Internet, (ii) replicas can fail, e.g., by crashing, the very events that have to be tolerated through replication, and finally (iii) the solution has to be deterministic as distributed algorithms generally are. Therefore, total-order protocols are developed by avoiding one or more of these three factors by resorting to realistic assumptions based on system contexts. Nevertheless, they tend to be complex in structure and impose time overhead with potentials to slow down the performance of replicated services themselves.

This thesis work develops an efficient total-order protocol by leveraging the emergence of cluster computing. It assumes that a server replica is not a stand-alone computer but is a part of a cluster from which it can enlist the cooperation of some of its peers for solving the total-order problem locally. The local solution is then globalised with replicas spread over a wide-area network. This two-staged solution is highly scalable and is experimentally demonstrated to have a smaller performance overhead than a single-stage solution applied directly over a wide-area network. The local solution is derived from an existing, multi-coordinator protocol, *Mencius*, which is known to have the best performance. Through a careful analysis, the derivation modifies some aspects of *Mencius* for further performance improvements while retaining the best aspects.

Contents

List of Notations and Abbreviations

n	number of nodes in the replicated system
f	fault-tolerance degree (maximum number of failures) of the replicated system
φ	degree of replicas within fail-signal process
FS_i	i^{th} Fail-Signal node of the replicated system
n_i	i^{th} node of the replicated system
p_i	i^{th} process hosted on Ni (executing protocol)
q	One of the i^{th} processes
m	message used for communication between two processes
rnd#	proposal round number for Paxos
rnd [q]	the highest-numbered round in which process q has participated in classic Paxos.
vrnd [q]	the highest-numbered round in which process q has *ACK* an order in classic Paxos.
vval [q]	the value process q has accepted in round *vrnd [q]* in classic Paxos.
lrnd [l]	the highest-numbered round that process l has started in classic Paxos
lval [l]	the value process l has proposed for round *lrnd [l]* in classic Paxos.
PREPARE	a message sent by the leader to acceptors in phase 1 in classic Paxos and Mencius.
PROPOSE	a message sent by the leader to acceptors in phase 2 in classic Paxos and Mencius.
ACCEPT	a message sent by the leader to acceptors in classic Paxos.
SUCCEED	a message sent by the leader to acceptors in classic Paxos.
ACK	a message sent by the acceptors to leader in classic Paxos and .
NACK	a message sent by the acceptors to leader.
LEARN	a message sent by the leader to all in Mencius.
BFT	Byzantine Fault-Tolerant order protocol.
v_i	value chosen by process p_i
l	leader process
r, r'	round number
I_{Pi}	is the index or sequence number of server P_i for the next simple

	consensus instance.
I_{FS}	is the index or sequence number of FS node for the next consensus instance.
Π	set of all processes or FS nodes
$\perp$	bottom value used in *Randomized* protocols
est_i	the estimated value of a process
val_i	a value picked by a process from a list of values
$\Diamond S$	simple of Failure detector oracle in Chandra and Toueg algorithm
$\boldsymbol{C_{Pj}}$	is the smallest instance that was not learned by $\boldsymbol{P_i}$ and has been coordinated by $\boldsymbol{P_j}$
β	defines the interval of instances that should be revoked in advance
NRM	the number of real messages
NSM	the number of skip messages
seq	request sequence number
req	client request
$\Diamond S$, Ω, $\Diamond P$	failure detector classes
α	a constant representing the accelerator outstanding messages
τ	a time constant for the accelerator to show that messages cannot be deferred for more than some time τ
S^{SI}	si^{th} site of the replicated system
SI	site index number
I_{SI}	Instance site index number
AI	request arrival time intervals
CAP	(C) Consistency, (A) Availability, and network (P) Partitions

Figures

Tables

Chapter 1
Introduction

Use of computers has become pervasive: from complicated space systems, medical instruments and military equipment to home appliances such as coffee makers and washing machines. With the advent of computer networks and the Internet, our lives are transformed and even influenced by our digital technology. Education systems, media, communication and business are now totally reliant on the computer and, in turn, are shaped by this technology in ways that we are only beginning to understand. We arrived at a point where we are attached and completely dependent on this machine.

Computer applications can be divided into two groups: non-critical applications verse critical applications. The disruption of any non-critical application has limited adverse effect and presents no danger to the health, safety, or security of individuals, and results no damage to the environment, or significant property damage. Nevertheless, for critical applications, the disruption and failure of such systems is expected to have a serious adverse effect, which could result in loss of life, or damage of property and the environment. Therefore, critical applications must guarantee dependability, which depends on the following factors: [KV93] reliability, availability, safety, and security.

- Reliability: the system can run continuously without failure.
- Availability: always ready to provide its services.
- Safety: nothing catastrophic happens, when the system temporarily fails to operate correctly.
- Security: confidentiality, intrusion tolerance

In areas where *availability* and *reliability* are the primary requirements of complex processing, the use of a single computer constitutes a single point of failure, due to hardware and software failure. There is an urgent need to overcome the problems due to single points of failure and this thesis will focus on the provision of uninterrupted computer service provisioning by reducing the reliance on single computing machine and by resorting to replication. Replicating service on multiple

servers that fail independently increases level of fault-tolerance and availability [SCH90]. Usually, replicas of a single server are executed on separate computers of a distributed system. The isolation of processors physically and electrically in a distributed system ensures that server failures are independent. Further, special protocols are employed to coordinate the interaction of clients with these replicas. A well-understood notion of client-replicas interaction is known as Client-Server paradigm. Client generates a request asking for a service, while server processes the request and sends back the response(s).

In order to achieve a higher degree of fault-tolerance and availability in distributed systems, two forms of computer redundancy have been proposed in the literature: first is called *primary backup* (or passive) replication [BMST93, BM92], and the second called *state machine replication* (or active) replication [GS97, SCH93, DGG05].

In *primary backup* approach one server is designated as the primary and others as backups. Only primary server is in charge of processing clients' requests. After processing a request, the primary server updates the state on the other (backup) servers and sends back the response to the client. If the primary crashes, one of the backups takes over, taking care to preserve the continuity in service state.

According to the way the primary responds to the client, we could have *blocking* or *non-blocking* primary-backup replication [BM92]. In the blocking primary-backup set-up, the primary sends its response to all backup servers and waits for an acknowledgement from all of them, during that time the client is blocked, this is considered as a potential performance problem; in the non-blocking set-up, the primary sends its response to the client without waiting for the acknowledgement from backups. This approach, be it blocking or non-blocking, is suited only to tolerate crash failures.

The *state machine replication* approach [SCH93] is more robust and can tolerate failures of types more serious than simple crashing. Here, the service is replicated on multiple servers and the responses produced by these servers are subjected to a majority vote. Thus, the incorrect or absent responses from failed servers are *masked* by the responses produced by the correct ones. This approach imposes *two requirements*. First, a service must be built as a deterministic state machine so that correct servers respond identically for the same request. Thus, the *first requirement* is concerned with the implementation of service software. The *second requirement*, on

the other hand, is concerned with replication management and can be stated as follows. At any timing instance *t*, let the sequences of requests processed by any two correct replicas until *t* be seq_1 and seq_2; for all *t*, either $seq_1 \subseteq seq_2$ or $seq_2 \subseteq seq_1$. In other words, if a correct replica processes any request *req* as the i^{th} request if and only if any other correct replica processes *req* as the i^{th} request. This second requirement will ensure that correct replicas produce identical responses for each request. Meeting this important requirement is generally referred to as the *Total Order* Problem [UHS+04, LAM01].

This work will propose and evaluate solutions to solve the total order problem in the context of wide area networks. These solutions can be used for implementing *state machine replication* which, by its masking potentials, can assure un-interrupted service provisioning when failures do occur. In the literature, such solutions are called *atomic broadcast* protocols (see [CT96]) but this report would generically refer to them as *total-order* protocols.

1.1 System Contexts for Solving Total Order Problem

1.1.1 Failure context

Components in a replicated system are prone to failures; components are classified into servers (or computers) and the network that connects the servers to each other. A component fails when it does not meet its specification. We assume that the network failures are masked through traditional means such as error detection and packet re-transmissions and that the specification of a network does not impose timing constraints for message delivery (more details in the next sub-section). Server failures can be classified into several schemas; the following are the two main models:

1- Crash failure: a server fails only by crashing, i.e., by halting, after which no output is generated by the server. Before crashing, it works correctly and generates its outputs according to its specification.

2- Byzantine failure [PSL80, LSP82]: this model of failure is the most serious one; in such environment a server may produce arbitrary response at arbitrary times. A server could generate an output it should never have generated, which cannot be detected as being incorrect.

We will not consider Byzantine failure model, this work will be focusing on crash failure model. This means that the responses produced by the replica servers need not

be voted on, but simply be subjected to identifying and discarding the duplicate responses.

1.1.2 Network Context

A distributed system consists of a finite number of servers interconnected through a communication network. The underling network that connects the servers can be either a *synchronous* network or an *asynchronous* network. In a synchronous network, there is a known, fixed upper bound on the time required for a message to be sent from one server to another and a known fixed upper bound on the relative speeds of different servers. In an asynchronous network there are finite bounds on communication delays but these bounds cannot be known.

The attraction and the interest of the asynchronous model come from its practicability. Such systems are characterized by public, wide-area networks, such as the Internet, and also by the local-area networks subject to unpredictable loads, such as those within clusters and data centres. The unpredictability of message transfer delays and process scheduling delays in those systems makes the asynchronous model a very general one.

Throughout this work, we will be assuming an asynchronous network. Within the class of asynchronous networks, we will distinguish between the local-area networks used within clusters and the wide-area networks (such the public networks) used to interconnect computers that are geographically wide apart. This distinction is motivated by the fact that message delays on wide-area networks can be considerably longer, possibly several orders of magnitude larger, than those on the local-area networks. A literature review of time delays for both networks found the following:

1. For WAN delays, Mencius [MJM08] assumes that one-way link delay of WAN is 25ms, 50ms, and 100ms. In the following paper [JS08], it is assumed that the RTT link delay of WAN is 100ms or 200ms. The average one-way link delays of WAN taken from a real experiment [CR+09] are found to be 110ms between Newcastle and Frankfurt, 533ms between Newcastle and Moscow, and 577ms between Newcastle and Los Angeles.
2. For LAN delays, our experimental measurements indicated a one-way link delay of LAN of approximately 3ms. In the following paper [MF09], it is assumed that the one-way link delay is 2ms.

Hence, any performance-oriented design of distributed protocols must attempt to limit the use of wide-area networks to be as minimum possible.

1.1.3 The Replicated System

The system is made up of several *sites* connected by a WAN, e.g., the Internet. Each site is a cluster of computers connected by multiple LANs. A given service is replicated in N sites, $N \geq 2$. Within each cluster site, the service is replicated on multiple computers which are called *servers*. The number of server replicas within a site must *ideally* be an odd number n that is larger than one, so that total ordering within that site can be done despite the crashing of a minority of these servers. A client can send its request over the internet to any one of the servers in any of these N sites. Typically, a client's request would be routed to the closest cluster site.

Note that the ideal requirement of n servers per site may be difficult to meet, if several services are to be replicated within a site. A way-out would be to actually replicate a service only on n', $1 \leq n' < n$, servers and use (n-n') *proxy* servers. The latter receive clients' requests for the service and cooperate with actual servers in ordering the requests, but do not process the ordered requests; instead, they receive the responses from the actual servers within the site and forward to the clients that submitted the requests to them. When $n' < n/2$, all actual servers within a site can crash; in that case, the proxies in that site need to receive the responses from actual server replicas in remote site or at least one of them should become an actual server. Throughout this report, we will assume $n'=n=3$ and $N=3$ and these numbers are chosen for easy comparison of performance with other related works in the literature.

In Figure 1.1 a replicated system that has 3 sites *($N = 3$)* is presented. Within each site, the actual service is replicated on 2 servers ($n' = 2$) plus one proxy, which makes *$n = 3$.*

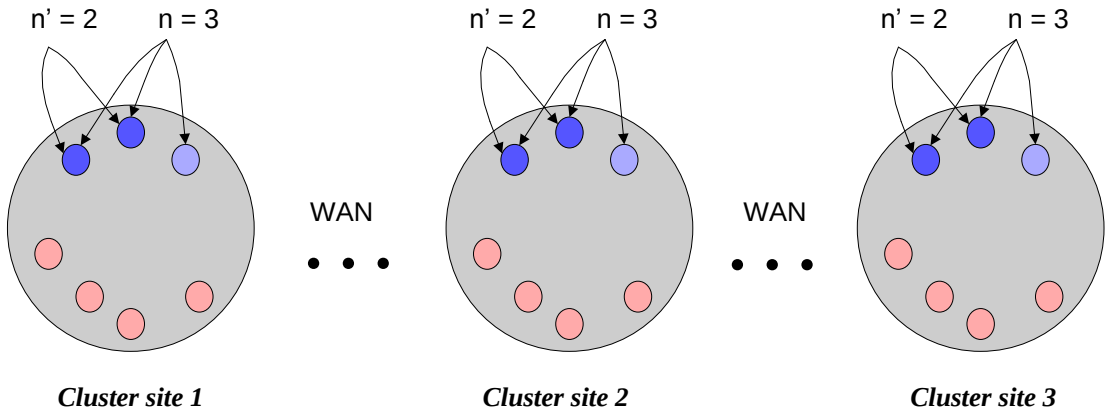

Figure 1.1. The Replicated System

1.2 System General Assumption

1.2.1 CAP Theorem

CAP (Consistency, Availability, and Partition) theorem [BRE12] has three properties**:** (1) state consistency (C), (2) availability (A), and (3) tolerance to network partitions (P) (asynchronous delays). CAP theorem states that distributed systems can preserve at most two of the three properties. Therefore, designers can choose the following desirable properties: only CA systems (consistent and highly available, but not partition-tolerant), CP systems (consistent and partition-tolerant, but not highly available), and AP (highly available and partition-tolerant, but not consistent) are possible.

So generally, in wide-area network, network partitions cannot be forfeit and a hard choice between consistency and availability remains. In line with CAP theorem in our thesis work, consistency and availability are preserved, however, network partitions (intolerant to network partitions) are forfeit. This is because consistency and availability can only be preserved when communication is possible. Thus, this implies that we assume there is no bound on wide-area network communication time delays.

1.2.2 Consistency vs. Latency

There is a tradeoff between consistency and latency [ABA12]. Choosing availability (small latency) over consistency in building a highly available system increases the complexity of distributed systems. State inconsistency demands higher level of design complexity in application development. Programmers must know when to use fast/inconsistent accesses versus slow/consistent accesses. The implication of the former approach might require the programmers to define the conflict resolution rules that meet the application's needs. To achieve small latency we have to sacrifice consistency, otherwise, preserving consistency forces us to pay the price in terms of large latency.

In this thesis work we present a protocol that solves *Total-Order* problem, securing high performance and correctness. This protocol guarantees consistency, while attempting to minimise latency.

1.3 Motivation and Challenges

The ever-increasing number of Internet-based e-commerce activities, the advent of clusters and Clouds, and the need for reliable and available services make the replicated system set-up presented earlier a *practical* and *scalable* one.

Note that the need for reliability and service availability has long been recognised and the service replication methods and the total order problem have also been well studied. In fact, as early as in 1978 Lamport [*Time, clocks, ordering.. CACM*] presented a solution for total order in a non-fault-tolerant environment. That solution itself required a quadratic message complexity and several fault-tolerant total-order protocols that followed are quite complex in structure and cannot avoid imposing a significant performance overhead.

What is new now is the volume of e-commerce activities involving client-server paradigm, the pattern of user access and the size of the client base; all of these emerging factors lead to the additional need for *scalable performance* of replicated services. The replicated system proposed in sub-section 1.1.3 is inherently a scalable one: *N* can be arbitrarily large and *n*' and *n* can be arbitrarily small (with the limit being 1 and 3 respectively). Such a system would be useless if the total-order protocol used were to impose a considerable overhead and thereby the overall client response time were to significantly slow down, even if processing an ordered request is to take

an insignificant amount of time. The challenge therefore is to develop a scalable and low-overhead protocol. This thesis work comprehensively and demonstrably addresses this challenge.

Understanding the causes of complexity and overhead associated with fault-tolerant total-ordering is best done by considering an equivalent, but widely analysed, problem of *distributed consensus*. Total-ordering and consensus are equivalent in the sense that a solution to one can be tailored as a solution for other; similarly, if one is not solvable in a given context, the other one also cannot be solved in that context [CT96].

The problem of consensus can be stated [FLP85] as follows. In a system of several failure-prone and distributed processes, each process has its own initial value; processes communicate with each other and reach an agreement on a common value subject to three conditions: (i) any two processes that decide must decide on the same value (*agreement*) (ii) the value decided must be any one of the initial values (validity) and all correct processes must decide at some point in time (*termination*).

Note that the total-ordering problem can also be equivalently stated: in a system of several failure-prone and distributed server replicas, each replica has its own initial preference for an order number indicating the order in which a request or a set of requests is to be processed; replicas communicate with each other and reach an agreement on a common processing order number, subject to three conditions: (i) any two replicas that decide must decide on the same order number (*agreement*) (ii) the order number decided must be any one of the initial preferences (*validity*) and all correct replicas must decide at some point in time (*termination*).

A major source of complexity and overhead associated with solving the consensus/total-order problem is due to the need to circumvent the FLP [FLP85] impossibility result: a deterministic protocol cannot be developed for an asynchronous network environment even if a single process can crash. This impossibility comes about because a slow process cannot be distinguished with total certainty from a crashed one when a bound on message transmission delays cannot be reliably established.

1.4 Approaches

Circumventing the FLP impossibility has been an active research area in past two decades. The most common approaches that have been proposed can be categorized into four types; multi-ordering protocols, deterministic protocols, randomised protocols and fail-signal protocols.

Randomized protocols [EMR01, MNC+06] are a family of protocols where FLP result is avoided by providing a probabilistic solution. Participants go over rounds of communication and make random choices on their estimate of decision values. The protocol progresses in such a way that eventually an identical value is decided. These protocols guarantee termination only in probabilistic terms which tend to 1 as elapsed time approaches infinity. This type of protocol is a non-leader protocol (where all nodes have the same quality, the same responsibility and have no use of unreliable Failure Detectors. Such protocols eliminate the need for detection and recovery from crash which is not an easy task because of the mistakes that can be produced by FD's.); however, the main disadvantage of this type of protocol is that the number of messages needed for termination is unknown, and the time needed to arrive to a decision my approaches infinity.

The second is called *Fail-Signal* protocol. Fail-Signal [BES+96, IE06] protocol is the third in the family of inherently redundant processes; namely, fail-stop and fail-silent processes, all of these three protocols are constructed in a similar way. FS is a protocol whose termination guarantee is not dependant on any systemic/network conditions and the performance is only affected by existing communication delays and real failures. Fail-Signal process circumvents the impossibility by making the failing process announce its imminent failure and stop working after failing. The main advantage of this type of protocols is the use of perfect failure detector. However, the main disadvantage of this approach is that each FS node consists of at least two machines connected by a synchronous network. This will result in a higher level of message complexity because all constituents of FS node will generate their own messages. For example, if FS node has two machines, then 2 identical proposals will be sent out to all correct FS nodes. Two identical *ack* messages will be sent out to the other FS node and 2 identical Learn messages will be sent out to all correct FS nodes (message redundancy). Another disadvantage is the high latency that results from waiting for the response from all processes.

The third type is called Deterministic protocols are built on the concept of Unreliable Failure Detector [CT91, CT96, CHT96]. Each process accesses *Failure-Detector oracle*, which provides a list of processes suspected to be crashed. The weakest form of Failure-Detector is denoted by ◊*S*, which allows it to solve consensus. This type of FD has the following properties: (1) any crashed process is eventually suspected (completeness), (2) there is a time after which correct processes are not suspected (eventual weak accuracy). This category of protocols tends to be coordinator-based. A specific process is given the role of coordination of the execution of the protocol, when it is crashed then the protocol chooses another process to play this role. Chandra Toueg [CT91] is considered to be the pioneer in this group. Paxos [LAM98] is well known protocol, which considered as one of deterministic approaches as well. Comparing to the other two categories, deterministic approach is characterized with lower latency and lower level of message complexity as well.

The last type is multi-ordering protocols. We consider *Mencius* [MJM08] as a novel and a new protocol belongs to this group. *Mencius* is a replicated state machine built on the abstraction of Paxos, it runs concurrent instances of Paxos. *Mencius* as a multi-ordering protocol tackles the issue of single leader bottleneck inherited from Paxos. Paxos suffers from some drawbacks in terms of communication pattern, CPU processing capacity, and latency of learning the outcome. By tackling the problem of single leader, the throughput is increased under high client load and latency is lowered under low client load.

Investigating *Mencius* over wide-area network will be the main theme of this work. In its published version, *Mencius* assumes *n*'=*n*=1 and *N* is some odd number that is larger than 1. Figure 1.2 depicts an example of *Mencius* system that has *N = 3*, and *n'=n=1*.

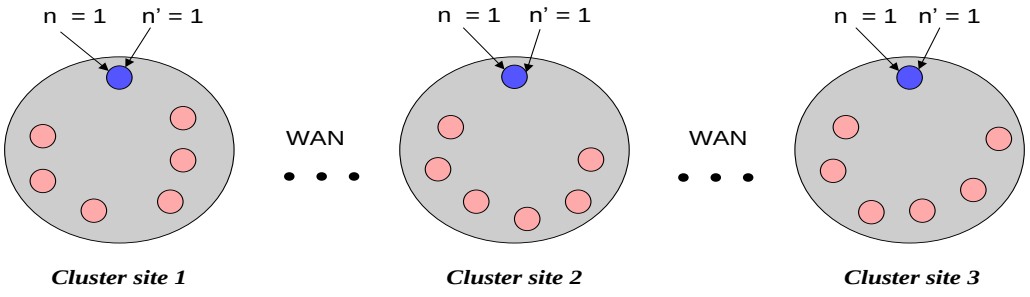

Figure 1.2 *Mencius*

This thesis work both qualitatively and experimentally establishes that *Mencius* can only offer sluggish performance when replicas are deployed across wide-area networks. We analyse its design aspects and propose a new variant which is shown to offer much better performance.

1.5 Contributions

Mencius [MJM08] was built over wide-area network and developed to tackle issues raised by single leader as in Paxos [LAM01]. In addressing these issues, it was able to achieve its objectives. Even with these achievements, we find that *Mencius* itself has its own issues that need to be addressed. *Mencius* has several problems that can substantially degrade its performance. These are revocation overhead, latency, false suspicion, crash, and bandwidth consumption of wide-area network.

We found out that the design of *Mencius* over wide-area network is the main source of all these problems. These problems are attributed to two issues: first, each site in *Mencius* has one server only. Second, Failure Detector reliability goes down on wide-area network. In order to overcome these problems, we decided to develop a new multi-ordering protocol called $[Mencius]^N$. The new proposed multi-ordering protocol will be a form of multi-cooperative *Mencius*.

In order to overcome the aforementioned problems, we decided to implement *Mencius* over a local-area network and build our protocol $[Mencius]^N$ on top of it. It is then distributed over wide-area network. Our solution will be presented in chapter 3 and chapter 4.

In chapter 3, we tackle the issue of revocation overhead in *Mencius* which is needed in case of false suspicion or crash.

In chapter 4, we present our second part of this work. The idea was to move the implementation of *Mencius* from a wide-area network to a local-area network instead. It is known that local-area network has higher band width, lower message delay time, lower latency, and lower rate of false suspicion occurrence compared to wide-area network.

Tackling these issues will produce a new protocol that has the following contributions:

1.5.1 False suspicion:

By moving the implementation of Mencius from wide-area network to a local-area network, the occurrence of false suspicion will be reduced.

1.5.2 Latency:

Our proposed solution will reduce the latency of client requests. This will be achieved by eliminating the execution of Paxos protocol over wide-area network.

1.5.3 Bandwidth consumption:

Reducing the number of exchanged messages over wide-area network between sites to finalize each instance. Unlike Mencius [MJM08], only one message is needed to report the outcome for each instance.

1.5.4 Threshold of saturation:

As each site consists of *n* nodes *(n = 2f + 1)*, there is the capability to cope with a higher rate of requests.

1.5.5 Removing client blocking:

Unlike Mencius, each site is built of *n* nodes *(n = 2f + 1)*, hence, as long as the majority is correct, no client will be ever blocked and that site will be able to order requests received from its group of clients.

These contributions lead to the following conclusions, which can be summarized on the following points:

1- Distributed application services that need to solve *Total Order* problem over local-area network are advised to use either *Mencius* or revised *Mencius*, as these two protocols tend to perform better than $[Mencius]^N$ over a local-area network.

2- However, for distributed application services that need to solve *Total Order* problem over a wide-area network, it is preferable to use $[Mencius]^N$, which is proved through the course of this thesis that $[Mencius]^N$ has better performance than *Mencius*.

3- It is possible to exploit message size in order to increase performance in $[Mencius]^N$. There is, however, a trade-off between throughput and latency in relation to message size. With a low request rate, it is suggested that single messages (no batching) is used. This produces better latency. However, if a high

request rate is used, users should resort to batching multiple messages in a single message in order to obtain better throughput.

1.6 Thesis structure

The structure of this work will be as follows:

Chapter 2 presents in detail the different approaches used to solve consensus. We will start by describing Fail-Signal approach [IE06], and randomized approach [EMR01]. Next, we will move to talk about deterministic approach, starting with Chandra and Toueg protocol [CHT96, CT96, CT91], then we will focus on Paxos and Mencius. Most of this chapter will be dedicated to the latter ones, as they form the foundation of our protocol.

Chapter 3 analyses *Mencius*, in particular, their claim that false suspicion and crash rarely occur. We believe that cases of false suspicion and crash occur frequently and the cost of revocation is very high. To minimize that cost, we made certain changes to Mencius protocol. Our modification is: we will revoke a whole range using one instance.

Chapter 4 presents our solution that fulfils the contributions mentioned above: $[Mencius]^N$ protocol built on top of *Mencius*. Basically, this protocol deals with crash failure model and adopts the same assumption adopted by *Mencius* which says that false suspicion and crash occurs rarely. The main concept of our new protocol is built on two levels: first, each site has n nodes $(n = 2f + 1)$. Second, the number of messages needed to finish each instance that is exchanged between sites over wide-area network will be reduced from 3 messages to only 1 message.

Chapter 5 presents our experiments and their results. Data measurements that are collected from our experiments will be used to analyse and compare the performance of both protocols. We will evaluate both of them according to their throughput and latency.

Chapter 6 presents the summary of this work, our conclusion and future work that could be carried out to continue in-depth research into the area of Multi-Ordering protocols.

Chapter 2

Related Work

2.1 Introduction

A common approach to achieve fault tolerance is to resort to *state machine replication* [GS97, DGG05, SCH93]. Commonly, the structure of client and server is used to design distributed systems. Normally, a service is implemented on one server and the client invokes the service by sending requests to that server. Using one server is the simplest way to provide the service; however, the level of fault tolerance is zero and unfortunately not acceptable. The provision of the service is available only as long as that server is functioning. In order to increase the availability, the service is replicated on more than one server built on separate physical processors that can fail independently. The method used to build such a system is *state machine replication*.

The most difficult challenge facing *state machine replication* is to keep all copies of the service in agreement and consistent. This is called *consensus*. Because of FLP [FLP85] impossibility result, consensus is not solvable by a deterministic algorithm in an asynchronous environment even if a single process crashes, because in such circumstances, we cannot distinguish between a slow process and a crashed one. Classic Paxos is one of these protocols that circumvents FLP and solves consensus when time restrictions are assumed remain valid for a sufficiently long time.

In the following sections we start by defining consensus and then present the four approaches used to circumvent FLP and solve consensus. The four approaches rely on what is called *oracle component* associated with each process. To solve the agreement problem, this component is acquired by each process to help in making a choice to reach agreement. There are three different oracles: First, *perfect Oracle* adopted by *Fail-Signal* [IE06] *approach*, its name indicating that this Oracle makes no mistakes. Secondly, *Random-Oracle* [EMR01], and *Suspector-Oracle* [DSU04]. This brief survey will be started by presenting Fail-Signal approach, randomized approach, then deterministic approach. In the last one, we will start by presenting Chandra Toueg protocol [CHT96, CT96, CT91] followed by classical Paxos. As our new protocol $[Mencius]^N$ is built on top of *Mencius*, and *Mencius* is built on top of Paxos, there is then a necessity to go into deep detail to describe Paxos. At the end of

this chapter, we will talk in detail about *Mencius* as well. Because the last two protocols form the foundation for our protocol, so the following lines will introduce both of them.

Paxos or classical Paxos is a protocol that executes infinite number of rounds. If in a round number $k = i$, a value v is chosen, then the protocol will guarantee consistency in a way that the same value v will be chosen in any round number $j > i$. Next we will consider Paxos in an environment where Fail Detector *FD-Oracle Ω* [CHT96] associated with each process to decide whether the local process should act as a leader or not, bearing in mind that *FD-Oracle Ω* is unreliable and can make a mistake. Then we will present a version of Paxos used to solve the *Total Order* problem. As one form of consensus, we will describe the work of the protocol where a group of clients sends requests, the leader will be responsible for ordering these requests, after arriving at an agreement with the majority of the processes that request will be decided.

At this point, we will introduce *Mencius* which will tackle some issues raised by the single leader, starting by explaining simple consensus protocol [MJM08], in which replicas take turns in proposing values (in contrast to Paxos, only the leader can propose values), In simple consensus, only one special replica (*coordinator)*, can propose any command; the others can only propose a special command *no-op*. At the end of this chapter *Mencius* protocol [MJM08] will be presented, which runs concurrent instances of simple consensus. But first we will state consensus.

2.2 Consensus

We assume that a distributed system consists of finite set of n processes, and for achieving consensus (or Total Order)no more than f processes can fail by crashing at any time, where $f \leq (n-1)/2$. Crash failure model means that a process either functions according to its specification or halts when it is crashed. Processes communicate by sending messages through an asynchronous network, where there are neither bounds on message delays nor on process speeds. The consensus problem is defined on a set of $\{p_1,p_2,\ldots,p_n\}$ processes participating in proposing and choosing a value. Each process p_i proposes a value v_i, and only a single value of the proposed ones can be chosen by all correct processes, not less than $f+1$. Protocols that solve consensus should guarantee:

(i) **Validity or nontriviality (Integrity):** only one of the proposed values can be chosen.

(ii) **Termination:** every correct process must decide.

(iii) **Uniform Agreement:** No two processes (correct or faulty) decide differently.

The uniform agreement does require even the faulty processes to decide identically with the correct ones, which is harder to achieve [CS00].

Safety will be ensured by (i) and (iii) properties: the state of all processes will be consistent. The second property ensures liveness, that the service processes continue execution and keep producing outputs. It is important to notice that, unlike agreement, uniform agreement is harder to achieve [CS00], as it is required that even the faulty process must not decide differently than the correct ones.

Note: whenever *f* is mentioned in this work it refers to the level of redundancy required for achieving consensus (or Total Order).

2.3 Fail-signal

The system is modelled as *n* FS nodes interconnected by a wide-area network figure 2.1. The FS node can be found in one of two states, as shown in figure 2.2, and the transition from one state to another can occur at arbitrary instants of time. The FS process behaviour in each of the states however is well defined and is explained in detail below.

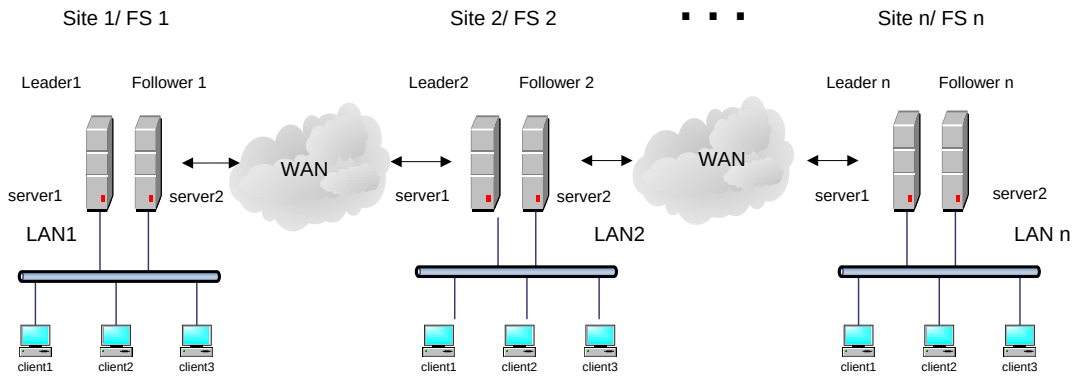

Figure 2.1: Network context of FS

- **Working State:** The Process is working correctly and free of faults, then all expected outputs are produced and each output produced is correct and sent to all relevant destinations. The FS process operates as per the specification of the program executed by its constituent process replicas, assuming that FS process at initialization found in this working state.

- **Signalled State:** The Process has halted at this terminal state; it emits only fail-signal to any destination to which an output is due; the fail-signal is uniquely attributable to the Process and cannot be undetectable forged by another Process.

Each FS node has two servers *(φ+1), φ = 1*, one called *Leader* and the other called *Follower*, and a group of clients. All these run on separate processors and communicate through a local-area network. FS nodes communicate with each other through wide-area network to implement a replicated state machine. Clients access the service by sending requests to their local servers via local-area communication. One of the FS nodes will act as a coordinator and the others will be considered as non-coordinators.

A Fail-Signal process should be implemented using (φ+1), φ ≥ 1, replica processes that are fail-independent, hosted on distinct nodes connected using a synchronous network. These replica processes (φ+1) are referred to as Fail-Signal (FS) node. Only one of these replicas can fail by crash at any time.

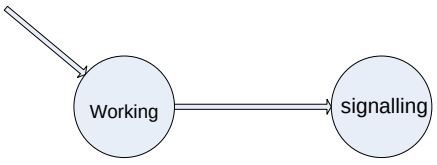

Figure 2.2: The two states of FS process

The mechanisms for constructing an FS process have been fully detailed in [BES+96, IE06]. The following will be a brief presentation of Fail-Signal model as detailed in

[IE06] with two states only. The FS process can be found in one of two states, as shown in Figure 2.2, and the transition from one state to another can occur at arbitrary instants of time.

One replica of FS node is called *Leader* and the other one is called *Follower*. The central idea is that both processes of FS node are engaged in active replication; they compare the outcome at each step of the execution of the protocol and relative timeliness of the counterpart. If both processes are correct and working according to their specification, then there is an agreement and the results are produced within timeliness. If one of the replicas is not correct (not working according to its specification), then a special signal pre-prepared sent out to all correct FS nodes informing them about the failure, and the protocol halts by transiting from Working state to Signalling state. Every FS node keeps a list of all crashed FS nodes that have already announced their failure.

The execution of the protocol starts by sending a proposal from the coordinator to all correct FS nodes. All correct FS nodes will respond by sending ACK message. Then the coordinator after receiving ACK from all correct FS nodes will send learn message to all correct FS nodes. It is worth to mention here that, there is a redundancy of messages produced and sent out by both constituent of each FS node. One of the main disadvantages of this protocol is its high level of message complexity.

2.4 Randomized Approach

The main concept of this approach relies on abandoning deterministic guaranty and providing a probabilistic solution to consensus. Hence the number of steps or messages needed to arrive at a decision is unknown. The protocol will decide a value with probability 1 when time *t* approaches infinity. In such a protocol, each process is accompanied with a component called R-Oracle, that generates a random value $x \epsilon \{1, \ldots, n\}$ when queried by the associated process.

Generally speaking, the protocol executes according to the following steps [EMR01]: the protocol goes through a number of rounds, each has two phases. In the first phase, process p_i proposes its value v_i to all other processes. This process has a variable that represents its estimate value est_i, initially equal to v_i. In this phase, each process will be waiting to receive from the majority their values, so each process will maintain a list of initial values received from other processes $val_i\{1:n\}$. If all the

values received correspond to one value *v*, then est_i will be updated to *v*, otherwise est_i updated to $\perp$. In the second phase, process p_i multicast its updated estimate value est_i to all other processes. The action taken by each process will be carried out according to the values received from the majority in the second phase:

- If all messages from the majority correspondence to value $v \neq \perp$, then $est_i = v$, and decide *v*.
- If at least one message is $v \neq \perp$, then $est_i = v$ and go to the next round.
- If all messages are $\perp$, pickup randomly one value from $val_i\{1:n\}$ and set est_i to that value and go to the next round.

By executing consecutive rounds of the protocol, the variable est_i will be updated to one of the values $val_i\{1:n\}$ proposed by one of the processes. This guaranties safety. The decision taken by all processes will converge to a single value which guaranty liveness. This type of protocol is a non-leader protocol; however, the main disadvantage of this protocol is that the number of message needed for termination is unknown.

There are a number of works that tolerate crash failure [BEN83, EMR01, CMS89], and others that deal with Byzantine ones [CD89, AH90, FM97, KS01]. As our work is concerned with deterministic approach, we will go no further in exploring randomized approach. We refer interested readers to an extensive survey of this approach on [ASP03].

2.5 Deterministic approaches

As our work is based on the deterministic approach, in this section we will start by investigating Chandra and Toueg protocol [CT91, CT96], then we will go deeper in exploring Paxos [LAM01, LAMO6], which forms the underlying protocol used by *Mencius*.

2.5.1 Chandra and Toueg protocol [CT]

The work presented by Chandra and Toueg [CT91, CT96] introduced the concept of Failure Detectors to tolerate crash failures. A Failure Detector is a component associated with each process. When queried, it provides information about the state (crashed or not) of other processes. There are several classes of failure detectors [CHT96]. One can recall that because of [FLP06], it is impossible to

correctly decide the state of other processes, whether they are alive or already crashed. In this work [CT96] they presented a range of failure detectors which can be classified according to two *completeness (being correct)* properties and four *accuracy (being wrong)* properties:

Completeness

We consider two types of completeness 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.

However, completeness by itself is not a useful property: a failure detector may trivially satisfy this property by always suspecting all the processes in the system. To preclude such behaviour, a failure detector must also satisfy an accuracy requirement that restricts the mistakes that a failure detector can make. Sections 3.2 and 3.3 consider accuracy and eventual accuracy, respectively.

Accuracy

We define two types of accuracy properties:

- Strong accuracy: Correct processes are never suspected.
  Since it is difficult (if not impossible) to achieve strong accuracy, we also define:
- Weak accuracy: Some correct process is never suspected.

The following three classes of failure detectors are defined:

- $\mathcal{P}$, the set of Perfect Failure Detectors that satisfy the strong completeness and the strong accuracy properties.
- S, the set of Strong Failure Detectors that satisfy the strong completeness and the weak accuracy properties,
- $\mathcal{W}$, the set of Weak Failure Detectors that satisfy the weak completeness and the weak accuracy properties.

Note that $\mathcal{P} \subset S \subset \mathcal{W}$ [CT91].

Even weak failure detectors guarantee that there is at least one correct process that is never suspected. Since this type of accuracy may be difficult to achieve, we consider weaker forms of accuracy in the following section.

Eventual Accuracy

The following types of failure detectors are considered that may suspect every process at one time or another. Informally, we only require that some accuracy property is eventually satisfied. We consider the following two types of eventual accuracy.

Eventual strong accuracy: There is a time after which correct processes are not suspected.

Eventual weak accuracy: There is a time after which some correct process is not suspected.

Each one of the three types of failure detectors that we defined in the previous section, we can replace the accuracy requirement with the corresponding eventual accuracy requirement. This results in the following three classes of failure detectors:

- $\Diamond\mathcal{P}$, the set of Eventually Perfect Failure Detectors that satisfy the strong completeness and the eventual strong accuracy properties.
- $\Diamond S$, the set of Eventually Strong Failure Detectors that satisfy the strong completeness and the eventual weak accuracy properties.
- $\Diamond\mathcal{W}^{o}$, the set of Eventually Weak Failure Detectors that satisfy the weak completeness and the eventual weak accuracy properties.

Clearly, $\Diamond\mathcal{P} \subset \Diamond S \subset \Diamond\mathcal{W}^{o}$, and $\mathcal{P} \subset \Diamond\mathcal{P}$, $S \subset \Diamond S$, $\mathcal{W}^{o} \subset \Diamond\mathcal{W}^{o}$ [CT91].

In [CHT96] they defined a new failure detector, denoted Ω, that is at least as strong as $\mathcal{W}^{o}$. They then show that any failure detector $\mathcal{D}$ that can be used to solve Consensus is at least as strong as Ω. Thus, $\mathcal{D}$ is at least as strong as $\mathcal{W}^{o}$. The output of the failure detector module of Ω at a process *p* is a single process, *q*, that *p* currently considers to be correct; we say that *p* trusts *q*. In this case we say that all failure detectors satisfy the following property:

There is a time after which all the correct processes always trust the same correct process:

As with $\mathcal{W}$, the output of the failure detector module of Ω at a process *p* may change with time, that is, *p* may trust different processes at different times. Furthermore, at any given time *t*, processes *p* and *q* may trust different processes.

2.5.2 Paxos algorithm

Paxos is a single leader protocol [BOI01], figure 2.3 shows network context of Paxos. The leader is the coordinator of the protocol and will be in charge of choosing the value. Paxos will be executed by *n* processes; one process will be elected as the leader and the others as non-leaders participating in the execution of the protocol. The leader will play a central role in taking the decision, and communication will be directed to the leader.

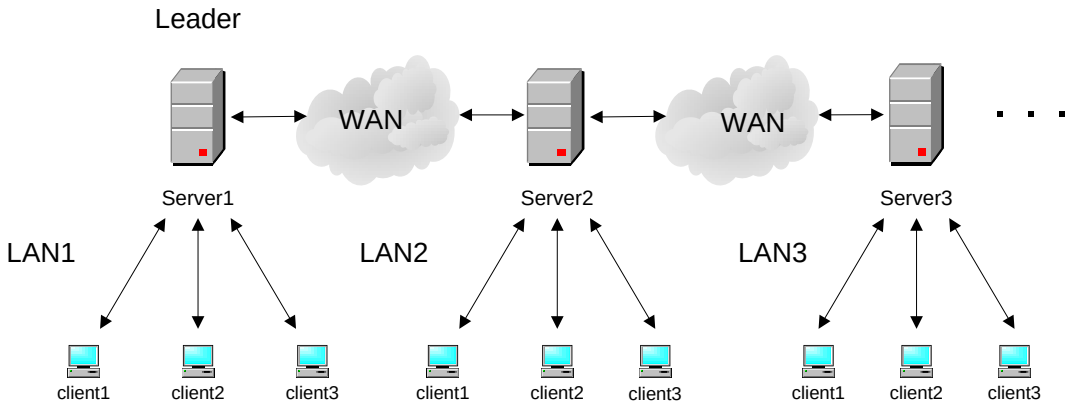

Figure 2.3: Single leader system

Paxos distinguishes three different roles played by each process, a single process can play more than one role: *proposer* is the process that propose a value that could be decided, *acceptor* is one of the processes that participate in taking decision, *learner* is the process that get informed about the decision which has been chosen. So we will

restrict ourselves to the use of these three terms (*proposer, learner, and acceptor*) to reference any process or server when describing Paxos.

In a Client / Server system, a client might play the roles of *proposer* and *learner*, and server might play the roles of *acceptor* and *learner*. All the *acceptors* participate in executing the protocol so long as they are correct, one of the *acceptors* play a distinguished role, which is called *leader.* Assuming that each server has a group of clients, and each client proposes its value to its server. If that server is not the *leader*, then the value will be forwarded to the *leader*, which will execute the protocol to choose a value with the participation of other correct *acceptors*, which should be ≥ *f+1*. The chosen value will be sent to the *learners*. *f+1* forms the majority of *acceptors* including the *leader* himself, this condition will guaranty consistency within all servers, because any two majorities will have at least one *acceptor* in common.

Example 1 Example 2

Figure 2.4 Client/Server role

Figure 2.4 presents two examples that depict the role that can be played by both the server and the client. These usually depend on the application. Example 1 shows servers playing the role of *acceptors* and *learners*. Servers take the decision and send the result to the clients. In our implementation in this work, we will be following the version presented in example 1, which is more useful in *Total Order*. Example 2

presents another aspect where servers act as *acceptors* and clients as *learners.* In this example, clients will participate in executing Paxos and will take the decision when they receive messages from the majority of the *acceptors*.

2.5.2.1 Paxos: no failure case

Here we present classical Paxos protocol [LAM06]. The protocol executes infinite number of rounds, and if in a round number $k = i$ a value v is chosen, then the protocol will guarantee consistency in a way that the same value v will be chosen in any round number $j > i$. Each round with a single *leader* that coordinates the execution of the protocol figure 2.5. One of the *acceptors* will play the role of the *leader*. Symbol l denotes the *leader* and symbol q denotes one of the ordinary *acceptors*.

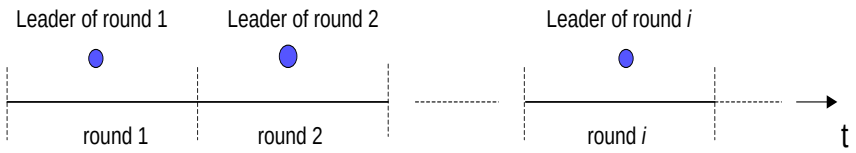

Figure 2.5 Paxos protocol executed in rounds

The following variables are maintained by *acceptor q*:

***Acceptor* variables :**

rnd[q] = 0 : the highest-numbered round in which *q* has participated.

vrnd[q] = 0 : the highest-numbered round in which *q* has *ACK* an order.

vval[q] : the value that *q* has accepted in round *vrnd[q]*.

The following variables are maintained by *leader l*:

***Leader* variables:**

lrnd[l] : the highest-numbered round that *l* has started.

lval[l] : the value that *l* has proposed for round *lrnd[l]*.

Messages used:

PREPARE, ACCEPT , SUCCEED - sent from leader to *acceptors*.

ACK, SUCCEED – sent from *acceptors* to leader or to all.

Figure 2.6 shows that the protocol executed in two phases. In phase one, the *leader* collects the values from the correct *acceptors*, by sending *prepare* message and receiving *ACK* from the majority. In phase two, the leader will choose a value, this value will be the leaders value if no value was chosen before. Otherwise, the value with the highest round number reported by at least one of the *acceptors* will be chosen, then the decision will made and disseminated to all *learners*. The following will be the description of the two phases.

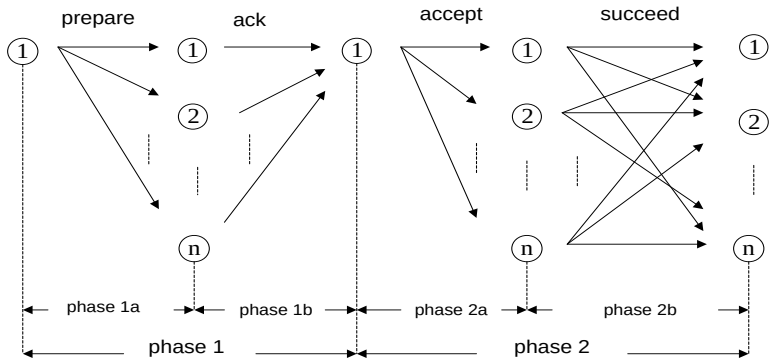

Figure 2.6: Classic Paxos

Phase 1

(a) - If *lrnd[l] < i* , then a new round *i* is started, *leader* sets *lrnd[l]* to *i* , setting *lval[l]* to *none*, and sending *PREPARE(lrnd[l] , lval[l]*) message to all *acceptors* asking them to participate in this round.

(b) - *Acceptors* will acknowledge *PREPARE* message *if lrnd[l] > rnd[q],* by setting *rnd[q] = lrnd[l],* then sending back *ACK(rnd[q], vrnd[q], vval[q])* the

acknowledgement message will be sent back to the *leader*. However, *if lrnd [l]* ≤ *rnd[q]*, the message is ignored.

Phase 2

(a) - the *leader* waits to receive *ACK(rnd[q], vrnd[q], vval[q])* message from the majority, if *lrnd[l] = i* and *lval[l] = none*, then *leader* has not begun a higher round, and has not performed phase 2a for this round. A value will be picked according to the following (1) if no *acceptor* has voted before, then *leader* picks any value *v*, (2) otherwise, pick value *v* of the highest round *vrnd[q]*. *Leader* sets *lval[l] = v*, and sends to all acceptors *ACCEPT(lval[l] , lrnd [l])* message to invite them to vote.

(b) - the *acceptor* receives *ACCEPT(lval[l] , lrnd [l]*) message from the *leader*, so *if lrnd[l]* ≥ *rnd[q]* and *vrnd[q]* ≠ *lrnd [l]* the *acceptor* sets *vval[q] = lval[l], vrnd[q] = lrnd [l], and rnd [q] = lrnd [l],* and a *SUCCEED* message sent to all. However if *lrnd [l] < rnd [q]* or *vrnd [q] = lrnd [l]* ignore the message.

2.5.2.2 Paxos: with Fail Detector case

Fail Detector requirements

Here we are considering Paxos in an environment where the leader may crash, and there is a need for leader change. FD-Oracle Ω associated with each process will decide whether the local process should act as a new leader or not, bearing in mind that FD-Oracle Ω is unreliable and can make mistake [CHT96]. This model of fail detector Ω guarantees that there is a time after which only one process is correct and not suspected by the other correct processes (Eventual Leader). Ω is used to elect a new leader when the current leader is suspected to be crashed. If a process considers itself a leader, then it will start a new round which should be higher than any previous round. If majority agrees to participate in this round, then a value can be decided. In some cases several processes may consider themselves leaders, but each will use a different round, the protocol guarantee progress until only one process with the highest round gets the majority and arrives to a decision and finally elected to be the new leader, so long as the majority working properly liveness is achieved. However; safety is ensured even when the election fails.

Presentation

The protocol as presented in figure 2.7 introduces a new message *NACK* [UHS+04] sent from acceptors to leader informing it that the round number should be higher than any round number received before from other leaders. The protocol is executed in three phases. In phase one, the *leader* collects the values from the correct *acceptors*, by sending *prepare* message tagged with the highest round number. The correct *acceptors* will respond by sending *ACK*, in case the message they have received was tagged with the highest round number. In phase two, if the *leader* receives a single *NACK*, then it will abort this round and will try again with a higher round number. On the other hand, if it receives *ACK* from the majority of the *acceptors*, a value will be chosen, this value will be the *leader's* value if no value was chosen before. Otherwise, the value with the highest round number reported by at least one of the *acceptors* will be chosen. In the third phase the decision will be made and disseminated to all *learners*. The condition for the protocol to make the transition from one phase to the next one is to receive the majority of *ACK* and no *NACK*. The following will be the description of the three phases.

Phase 1

(a) - If *lrnd*[*l*] < *i* , then a new round i is started, leader sets *lrnd*[*l*] to *i* , setting *lval*[*l*] to *none*, and sending *PREPARE(lrnd*[*l*] , *lval*[*l*]) message to all acceptors asking them to participate in this round, this message will be resent continuously until *ACK* message is eventually received from the majority, or even one *NACK* is received.

(b) - Acceptors will acknowledge *PREPARE* message *if lrnd[l] > rnd [q],* by setting *rnd [q] = lrnd [l],* then sending back *ACK(rnd [q], vrnd [q], vval [q])* the acknowledgement message will be resent continuously until the leader eventually receives it. However, *if lrnd [l] ≤ rnd [q],* negative acknowledgement *NACK* is sent. This is the case when we have more than one leader.

Phase 2

(a) - the leader waits to receive *ACK(rnd[q], vrnd[q], vval[q])* message from the majority, *if lrnd[l] = i and lval[l] = none,* then *leader* has not begun a higher round, and has not performed phase 2a for this round. A value will be picked according to the following (1) if no *acceptor* has voted before, then *leader* picks any

value *v*, (2) otherwise, pick the value *v* of the highest round *vrnd[q]*. *Leader* sets *lval[l] = v*, and sends to all *acceptors ACCEPT(lval[l] , lrnd[l])* message to invite them to vote. However, if a single *NACK* message is received then the *leader* aborts this round and tries later with higher proposal number.

(b) - the *acceptor* receives *ACCEPT(lval[l] , lrnd[l])* message from the *leader*, so *if lrnd[l]* $\geq$ *rnd[q]* and *vrnd[q]* $\neq$ *lrnd[l]* the *acceptor* sets *vrnd[q] = lval[l]* and *rnd[q] = lrnd[l]*, then sending back *ACK(rnd[q])* message, which will be resent continuously until the leader eventually receives it. However, *if lrnd[l]* $\leq$ *rnd[q]*, negative acknowledgement *NACK* is sent. This is the case when we have more than one leader.

Phase 3

(a) - the leader waits to receive *ACK(rnd[q])* message from the majority, *if lrnd[l] = i*, then the leader sends *SUCCEED* message to all. However, if a single *NACK* message is received then the leader aborts this round and tries later with higher proposal number.

(b) - the acceptor receives *SUCCEED* message from the leader, will decide.

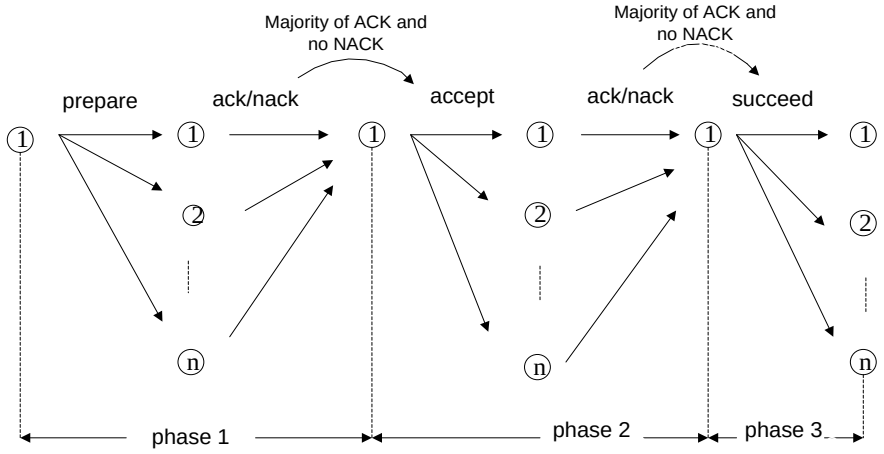

Figure 2.7: Paxos with three phases

2.5.2.3 Paxos for *Total Order* problem

Paxos protocol is used to solve *Total Order* problem as a form of consensus as presented in [UHS+04, LAM01]. [FLP85, DDS87] Show that *Total Order* can be transformed into consensus, and vice versa. Hence, the impossibility result holds for both problems. Therefore; any protocol solves *Total Order* problem must satisfy the following properties:

(i) **Termination** – If a correct process delivers *m*, then all correct processes eventually deliver *m*.

(ii) **Integrity –** For any message *m*, every correct process delivers *m* at most once and only if *m* was broadcasted by some process.

(iii) **Total Order**– If two processes (correct or faulty) *p*1 and *p*2 deliver messages *m*1 and *m*2 then *p*1 delivers *m*1 before *m*2, iff *p*2 delivers *m*1 before *m*2.

The protocol will be defined according to the following context: a set of all processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ participating in *Total Order* protocol, where the total number of processes $n \geq 2f+1$. Each process consists of a *learner* that represents the service and an *acceptor* that represents the ordering protocol. The ordering processes *acceptors* execute a protocol that ensures the above three *Total Order* broadcast properties. On receiving every new request from a client, *acceptors* communicate with each other to assign a unique and identical order number. Hence, all correct *acceptors* forward all clients' requests in identical order to the corresponding *learners* for execution. This leads to identical result generation at various replicas.

At this stage we will focus on a failure-free run, with fixed leader and no crash. Paxos protocol *Total Order* problem in two phases. First phase used for choosing the value, while the second phase used to commit the value. In case of *Total Order* problem on a failure-free run the first phase will be replaced by the client sending a request to the *leader*. In such case there is no need to implement the first phase which used to collect values from all correct *acceptors* to check whether they have participated in taken some decision in different rounds. Here we will present two versions of *Total Order* protocol. In the first version acceptors will send acknowledgement to the *leader* only after receiving accept message from the *leader*, then there is a need for a third phase as in figure 2.7, in which the *leader* will inform

all *acceptors* about his decision, where in the second version after receiving accept message from the *leader* acknowledgement is sent by the *acceptor* to the *leader* as well as to all *acceptors*, following from that each *acceptor* will be able to decide on its own after receiving *ACK* from the majority, in this case there is no need for the third phase.

Paxos can be implemented using either 2-phase protocol (Classic Paxos), as presented in section 2.5.2.5 and figure 2.9 or 3-phase protocol, as the one presented in section 2.5.2.4 and figure 2.9. Even though the latter has more phases than the former, the message complexity in the 2-phase version is $\{3(n-1)+n^2\}$ which is higher than the 3-phase $\{5(n-1)\}$.

2.5.2.4 Normal operation in failure-free situation ACK is sent to the *leader* only

Using client/ Server context, and assuming that the *leader* is not going to crash. We have one *leader* which will not be changed during the course of protocol execution. The client sends a request to the *leader*, then the *leader* with the participation of other correct *acceptors* execute the ordering protocol according to the following steps:

(a) - After receiving a request from the client, *leader* gives an order number to this request sends *ACCEPT* message for this order to all *acceptors*, this order number should be higher than all previous order numbers *ACCEPT(O_i , rnd#)*.

(b) – Following the receipt of *ACCEPT(O_i , rnd#)* message, *acceptors* acknowledge that by sending *ACK(O_i , rnd#)* message to the *leader*

(c) – As soon as the *leader* receives *ACK(O_i , rnd#)* from the majority of the *acceptors*, it will respond by sending *SUCCEED* message to all *acceptors*.

(d) – *Acceptors* after receiving *SUCCEED* message from the *leader* will decide.

Figure 2.8 shows the above mentioned steps, this figure is similar to the one depicted in figure 2.7, except that in the first phase of figure 2.8 the client sends a request to the *leader*, instead of *leader* sending *PREPARE* to *acceptors* to choose the value.

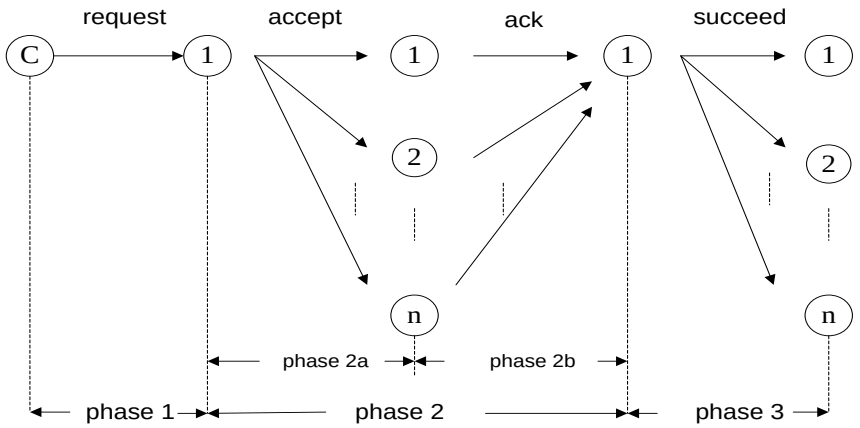

Figure 2.8: Paxos used to solve *Total Order*

2.5.2.5 Normal operation in failure-free situation ACK is sent to all

(a) - After receiving a request from the client, *leader* gives an order number to this request sends *ACCEPT* message for this order to all *acceptors*, this order number should be higher than all previous order numbers *ACCEPT(O_i , rnd#)*.

(b) – Following the receipt of *ACCEPT(O_i , rnd#)* message, *acceptors* acknowledge that by sending *ACK(O_i , rnd#)* message to the *leader* and all *acceptors*.

(c) – If the *leader* and *acceptors* receive *ACK(O_i , rnd#)* from the majority of the *acceptors*, they will decide on their own.

Figure 2.9 shows the above mentioned steps, this figure similar to the one depicted in figure 2.6, except that in the first phase of figure 2.9 the client sends a request to the *leader*, instead of *leader* sending *PREPARE* to *acceptors* to choose the value.

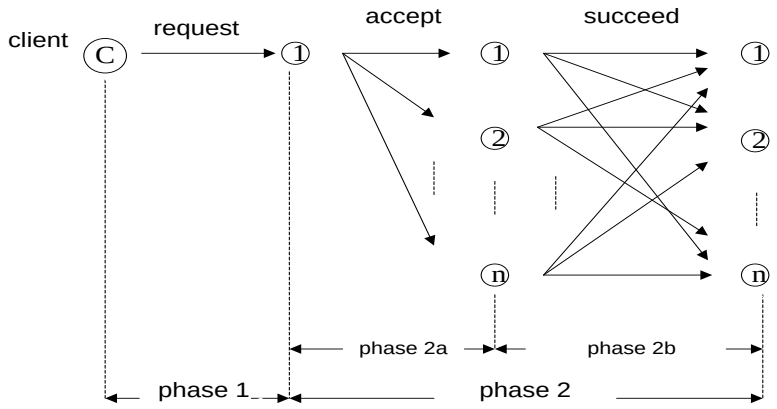

Figure 2.9: Paxos used to solve Total Order

2.5.2.6 *Total Order* Protocol with leader change

The execution of the protocol explained here will follow the example of Paxos presented at section 2.5.2.2 and figure 2.7. In case of crash or suspicion of failure of the existing *leader* one of the correct *acceptors* will be selected by its Ω to play the role of the new *leader*. The following steps explain the process of *leader* change:

Phase 1

(a) – when the new *leader* gets selected by its Ω, it will start executing first phase as shown in figure 2.10. The new *leader* needs to learn the history of other *acceptors* (order numbers that were decided). The *PREPARE* message will be tagged with a proposal number *rnd#* that is higher than any proposal number received from any predecessor *leader*, in addition to that it will carry a list of all missing/undecided orders. The purpose of this *PREPARE* message is to seek the highest proposal number less than *rnd#* for each of these order numbers that has been accepted by any *acceptor*.

(b) – An *acceptor* responds to *PREPARE* message by sending either *ACK* or *NACK* to the new *leader*. An *acceptor* will send *NACK* in case there are any order numbers with higher proposal number than *rnd#.* In this scenario *NACK* message could be made to carry the proposal number higher than *rnd#* corresponding to each order number for which an accept message was sent. This will guide the new leader which can attempt again with a new large enough proposal number. On the other hand; if the *rnd#* is higher than any proposal number seen by this *acceptor, ACK* message sent back to the new *leader*, and it must contain all order numbers and their corresponding requests for those reported missing/undecided by the new *leader* in *PREPARE* message if any was accepted locally.

Phase 2

(a) – after receiving *ACK* responses from the majority, the new *leader* will start executing phase two. It will send *ACCEPT* message carrying all order numbers up to the highest one reported. For unreported requests *no-op* will be sent to fill in the gap. If *NACK* was received the new leader aborts this round and tries later with a higher proposal number higher *rnd#*.

(b) – Following the receipt of *ACCEPT*(O_i , *rnd#)* message, *acceptors* acknowledge that by sending *ACK*(O_i , *rnd#)* message to the *leader*, if they have not acknowledged *PREPARE* message with a higher proposal number *rnd#*, otherwise *NACK* message will be sent back to the leader. In case of a single leader, this is always will result in sending back *ACK*(O_i , *rnd#)* message.

Phase 3

(c) – If the *leader* receive *ACK*(O_i , *rnd#)* from the majority of the *acceptors*, it will decide and send it to all correct *acceptors*. However, if the *leader* receives even a single *NACK* message, it will retry later by sending a new proposal with higher proposal number.

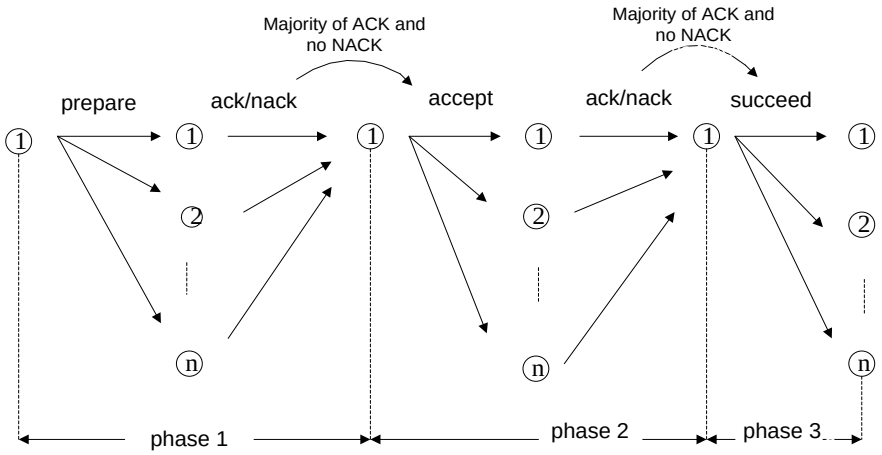

Figure 2.10: Paxos used to solve Total Order

An Example of Leader change

An example quoted from [LAM01] to elaborate on leader change. Suppose that a new leader has been selected by Ω. Say the new leader knows about order assignments 1-134, 138 and 139. It will execute part 2 for orders 135-137 and all orders above 139 to find out if these are accepted by any process. It can use a single sufficiently higher proposal number *rnd#_1* from its pool to construct a prepare message for all these missing orders as mentioned in (a) of Part 2 above. Suppose it received ACKs from at least a majority of processes containing requests with order numbers 135 and 140 only. Since at least a majority has not accepted orders 136, 137 and higher than 140, these orders could not have been decided. To fill in the gaps, the new leader proposes 136 and 137 to be assigned to no-op requests. Hence the new leader can start executing part 1 of the protocol for every new request with 141 to be the first proposed order number.

Another possibility is that the new leader receives a NACK containing a proposal number *rnd#'*, *rnd#' > rnd#_1*. This implies that the sender process has accepted at least one of these missing orders for *rnd#'*. Therefore, the new leader chooses a proposal number *rnd#_2* from its pool such that *rnd#_2 > rnd#'* and restarts execution of part 2.

2.6 Deriving *Mencius*

We dedicated the previous sections of this chapter to define Paxos, but the rest of this chapter will concentrate on how *Mencius* was derived gradually. The authors of paper [MJM08], introduced the concept of simple consensus, which was built based on Paxos. Then they construct an intermediate protocol *P*, which runs an unbounded sequence of simple consensus. Protocol *P* was described using four rules, and finally by adding three optimizations and one accelerator they derived *Mencius*, figure 2.11. The benefit of gradual development of *Mencius* is that, they showed that simple consensus is correct, so *Mencius* is correct as well.

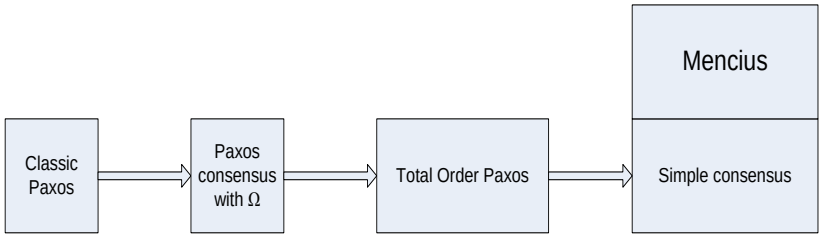

Figure 2.11: Deriving *Mencius*

Mencius is a replicated state machine that runs concurrent instances of simple consensus. The system has n sites *(n=2f+1)* interconnected by a wide-area network figure 2.12. Each site has a server and a group of clients. These run on separate processors and communicate through a local-area network. Servers communicate with each other through the wide-area network to implement a replicated state machine with 1-copy serializability consistency. Clients access the service by sending requests to their local server via local-area communication. We consider each site as a leader that orders requests received from the group of clients connected to it through local area network, so we have *n* leaders. When a server crashes, no request issued from its local area network will be ordered. Anyone of the correct servers can replace the crashed one to fill in the gap by producing no-op message.

The approach that will be taken in presenting *Mencius* in the following sections is that, we will highlight the bottleneck of single leader, because single leader problem is focal point that *Mencius* tackled. Next we will explore simple consensus, then we will move straightaway to present *Mencius* itself omitting protocol *P*.

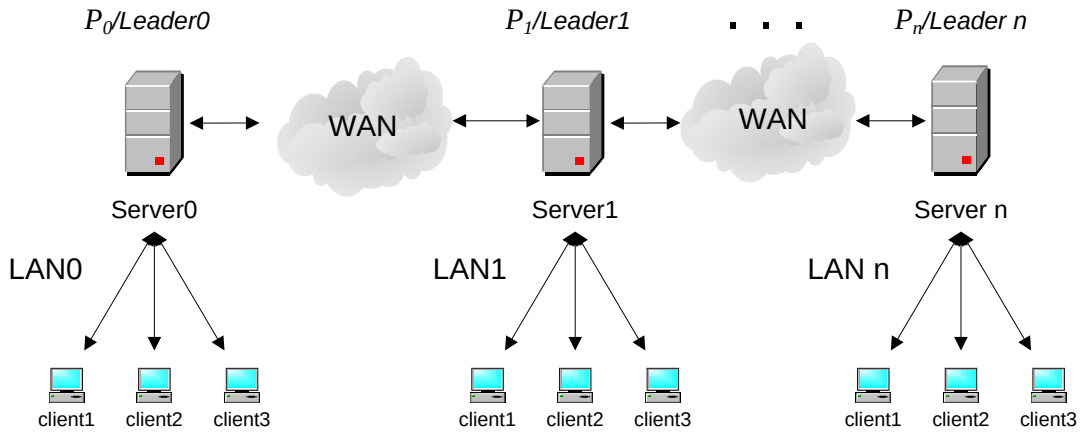

Figure 2.12: Network context of *Mencius*

2.6.1 Single leader bottleneck

Paxos is a single leader protocol like other single leader protocols suffers from some drawbacks in terms of communication pattern, CPU processing capacity, and latency of learning the outcome. Addressing these drawbacks *Mencius* succeeded in achieving high throughput under high client load and low latency under low client load. These problems will be highlighted, and the following sections will explain how *Mencius* addresses them.

Clients on the same site as the *leader* enjoys low latency (clients connected to server1) figure 2.3, because the outcome will be learned in two communication steps or messages (*propose, accept*). First step, proposing the request to other servers (*acceptors*). Second step, receiving accept message from majority, then the outcome is ready for the clients of the *leader's* site. However; clients on other sites have higher

latency, because they will suffer four communication steps or messages *(forward, propose, accept, succeed)*. First step, one of the *acceptors* forwards client's request to the *leader*. Second step, proposing the request to other servers (*acceptors*). Third step, receiving accept message from majority. The forth step, sending learn to all *learners* informing them about the outcome.

The second problem is the communication pattern, all messages will be propagated to the leader, while the channels between non-leader are idle, which will not utilize the available bandwidth of the whole system. This problem compounds when the system is *network-bound*, which means message size large enough to cause the channels to saturate before the CPU reach its limit. Therefore; shorter messages increase network bandwidth available to send more requests. The throughput of a *network-bound* system will be judged by how efficiently the message size is chosen.

The last one, the *leader* processes more messages then other replicas, because all requests are forwarded to the *leader* from other *acceptors*, especially when the system is *CPU-bound*. When the messages are of small size the number of requests received by the leader is increased and the *leader's* CPU is fully utilized, while the other replica are not, consequently, the total system processing power will never be utilized. *CPU-bound* system put more demand on the CPU processing power to cope with the high number of requests received. The throughput of a *CPU-bound* system will be judged by the CPU power capacity.

2.6.2 Simple Consensus

To derive multi ordering protocol *Mencius* [MJM08], simple consensus was built on top of classic Paxos, the concept behind that is in Paxos only the leader is allowed to propose values figure 2.13.

Values that can be proposed by simple consensus are restricted to two values only, either a value representing a client request or *no-op*. *No-op* is a value that makes no change to the state of the system, hence no response will be generated by the receivers of this value.

Phase	Classic Paxos 2-phase	Paxos with 3-phase	Simple consensus
Phase 1a	*Prepare*, From leader to all	*Prepare*, From leader to all	*Prepare*, From coordinator to all
Phase 1b	*Ack*, From each to leader	*Ack*, From each to leader	*Ack*, From each to coordinator
Phase 2a	*Accept*, From leader to all	*Accept*, From leader to all	*Propose*, (*suggest* or *no-op*) From coordinator to all
Phase 2b	*Succeed*, From each to all	*Ack*, From each to leader	*Accept*, From each to coordinator
Phase 3	No	*Succeed*, From leader to all	*Learn*, From coordinator to all

Table 2.1: Comparing message naming in each phase

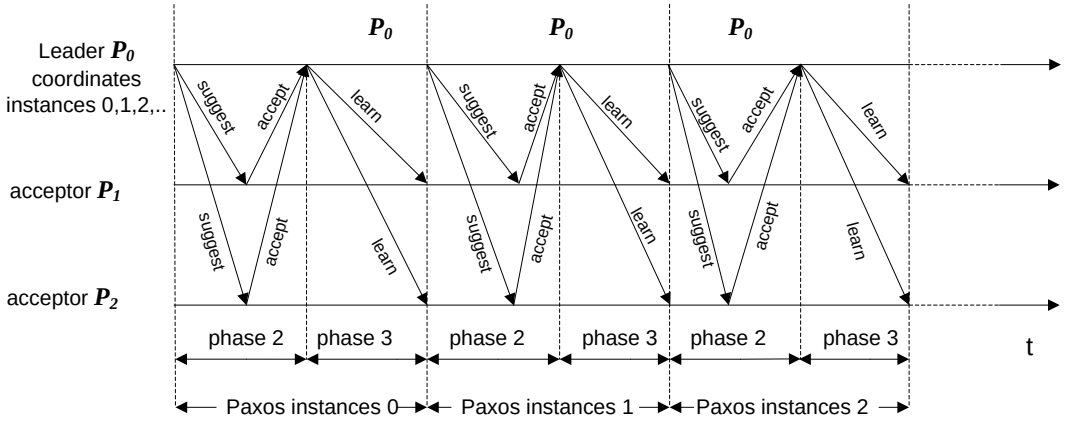

Figure 2.13: Instances of Paxos

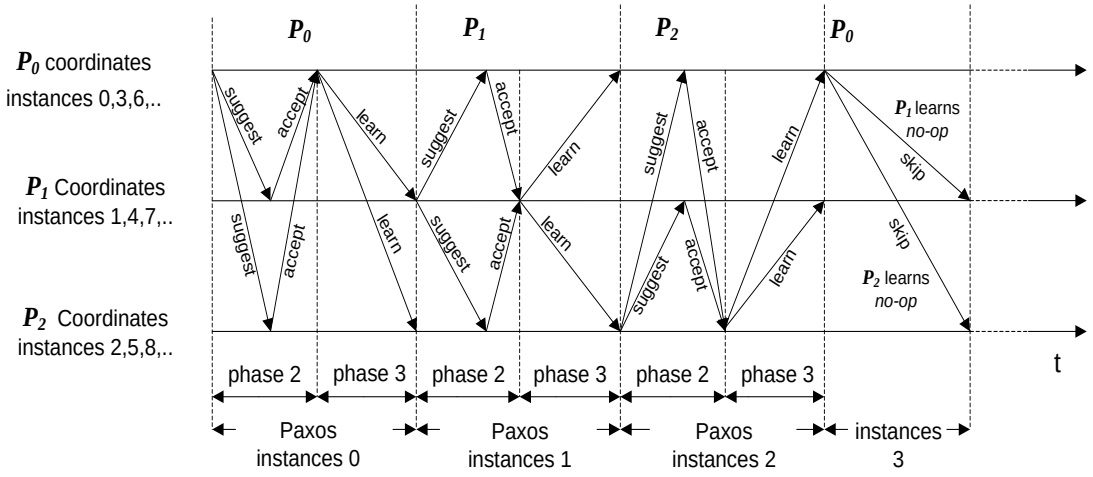

Figure 2.14: Instances of Simple Consensus

In simple consensus servers take turns in proposing values figure 2.14, only one special server (*coordinator)*, can propose any value (including *no-op*); while the others can only propose *no-op*. The benefit of using Paxos to implement simple consensus is that, Paxos is proved to be correct and guarantee safety and liveness, following from that simple consensus is correct. The terms used in this context will be different than the one used with Paxos. *Acceptor* and *Leader* will be replaced by *server* and *coordinator* respectively. Up to the end of this chapter we will reference each server that has turn to coordinate an instance *i*, as the *coordinator* of that instance and others as *servers*.

2.6.2.1 Assumptions and requirements

In addition to the original assumptions made about Paxos (crash model and Failure detector) the following one will be added: the underlying network connecting servers will be based on asynchronous FIFO communication channels. Since TCP is the underlying transport protocol, we assume messages between two correct servers are eventually delivered, and delivered in order.

Every server is the coordinator of an unbounded number of instances, for every server p there is a bounded number of instances assigned to other servers between consecutive instances that p coordinates.

$cn+p$ to server p, where $c \in \mathrm{N}_0$ and $P \in \{0, \ldots, n-1\}$.

For a system that has 3 servers n = 3.

" $P_0 = 0, 3, 6, \ldots$"

" $P_1 = 1, 4, 7, \ldots$"

" $P_2 = 2, 5, 8, \ldots$"

2.6.2.2 Messages sent by server and their actions

According to Table 2.1 messages used with simple consensus that might have different meaning or action are the following:

1. PREPARE: When the coordinator has been suspected to be failed, some server will arise eventually as the new leader and revoke the right of the suspected coordinator to propose a value. This is accomplished by finishing simple consensus instance on behalf of the suspected coordinator. As in Paxos the new

leader will execute first phase to learn if a value may have been chosen for some round $r' > r$, then this value will be proposed in phase 2. Otherwise, *no-op* will be proposed in phase 2. This operation is called *REVOKE*.

2. PROPOSE: When the coordinator needs to *suggest* a request *v*, then the payload of PROPOSE message is *v* in round *r*, this message will be called SUGGEST. On the other hand, when the coordinator wants to *skip* its turn, it sends *PROPOSE* messages with a payload of *no-op* in round *r*, this message will be called *SKIP*.

3. ACCEPT is used as a response for PROPOSE instead of ACK.

4. LEARN message is used instead of SUCCEED, to inform correct servers about the outcome of the consensus instance.

2.6.2.3 Simple consensus with no crash

Assuming that all servers have received requests ≠ *no-op* from their clients, they will start suggesting them. According to simple consensus *coordinator* P_0 has the turn to coordinate instance 0 by suggesting a request, while the other two *servers* P_1 and P_2 wait for the outcome of that instance. After learning the outcome P_1 will take the next move to be the coordinator of instance 1, and server P_0 and P_2 will be waiting for the outcome of that instance. When that instance concludes P_2 takes its turn to coordinate instance 2, and this instance will be accomplished in the same way as the others. This will continue as long as the servers have requests to suggest. In case a server has the turn to coordinate an instance with no request available to suggest, then it sends *SKIP* (*no-op)*, which has no effect on the system state and will be learned immediately.

The importance of *SKIP* is that, it will release other servers from waiting for their turn which may take a very long time. Each server has the turn to coordinate an instance *i* figure 2.14, either suggests a request or *SKIP* its turn in case no request is available.

It is worth at this point to differentiate between *learning* a value and *committing* a value in the light of simple consensus protocol.

The output of simple consensus protocol sent to the application service using *learn* message. Each server will produce its own instances independently, which may result

in delivering different instances from different servers in random order. However; the application service guaranty that those message will be *committed* (consumed) in the right order figure 2.15, which might lead to some delay in committing instances.

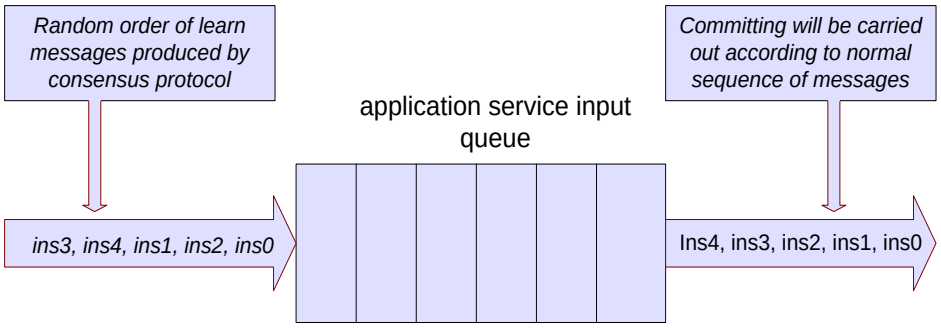

Figure 2.15: Committing instances in the right order

In case the application service did not learne *instance i-1*, but has already learned *instance k*, where $k \geq i$. then *instance k* cannot be committed till all *instances* $< k$ should be received.

The description given above emphasizes that as long as there is no crash, the ordering process proceeds smoothly. When sending *learn* message to the application service, it is assumed that everything is in order, and even though the instances are not delivered in sequence, sooner or later the other instances will be learned. But the difficult issue is how does the system respond to a crash? With crash, some instances will not be produced as long as the crash is there. Missing instances which should have been produced by the crashed server will prevent the system from making any progress. This is the reason behind differentiating between *learning* and *committing.* To overcome this problem we resort to *REVOKE* operation, which will be explained next.

2.6.2.4 Simple consensus with *REVOKE*

After describing how simple consensus behaves in a no crash environment, we will now focus our attention on how this protocol will react in circumstances when

one coordinator crashes. We have already mentioned that if a server has the turn to coordinate an instance *i,* it must do so by either suggesting a request or skipping its turn. This gives a chance for other servers to commit what they have already learned without being blocked for a long time. The new scenario is what is the consequence if a coordinator crashes? We assume that P_2 crashes (figure 2.16), and *no suggest*, or *no SKIP* will be produced any more by that server. The application service of the other two servers P_0 and P_1 will not be able to commit any new instances as long as P_2 crashed, which may go for ever. To solve this dilemma, one of servers P_0 or P_1 must raise up and coordinate instances on behalf of P_2. In our example server P_0 will replace P_2 in coordinating instance 2. P_0 will send *PREPARE* to all correct servers to check whether a value has been suggested or not, when the majority respond by sending *ACK* with no value the new coordinator will send *no-op* for this instance 2. Now all correct servers will learn *no-op* for this instance.

The protocol will go back to normal work as it was described early. If the same server P_2 still crashed, then for the next instance 5 that it should have been coordinated, will be revoked in the same way either by P_0 or P_1. This is how crash problem is overcome in simple consensus.

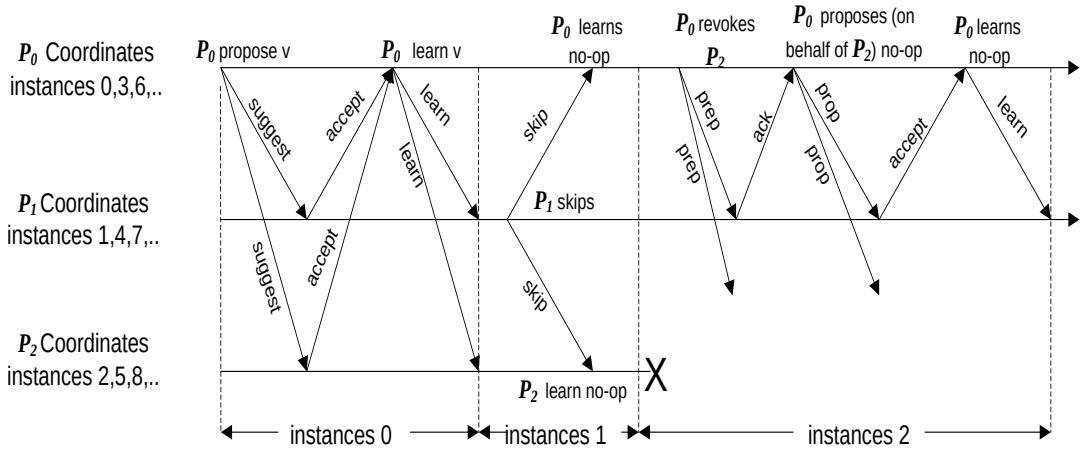

Figure 2.16: Simple consensus

2.7 *Mencius*

As we mentioned early that *Mencius* will be derived in terms of the following:

- four rules,
- three optimizations,
- and one accelerator.

Each server will run two services or processes. One service or process executes *Mencius* consensus protocol, while the other one executes the application service itself.

Mencius is a multi-ordering protocol, each server orders requests independently with the help of other servers, but committing these orders require tight coordination from other servers. This is inherited from the nature of *Mencius* where the sequence of consensus protocol instances is partitioned among the servers. The sequence is generated on the global level, which make servers tightly connected when it comes to committing the requests. The following sections will illustrate how this issue was tackled.

Rule1. Each server *p* maintains its index number *Ip*, for a system that has three servers P_0, P_1, and P_2 then $I_{P0} = 0,3,6,\ldots$; $I_{P1} = 1,4,7,\ldots$; *and* $I_{P2} = 2,5,8,\ldots$,a server *p* suggests the client's request to consensus instance *Ip* and updates *Ip* to the next instance. If the speed of suggesting values of all servers is at the same rate, then rule 1 is sufficient for good performance. Figure 2.17 reflects the ideal system, all servers working with the same speed.

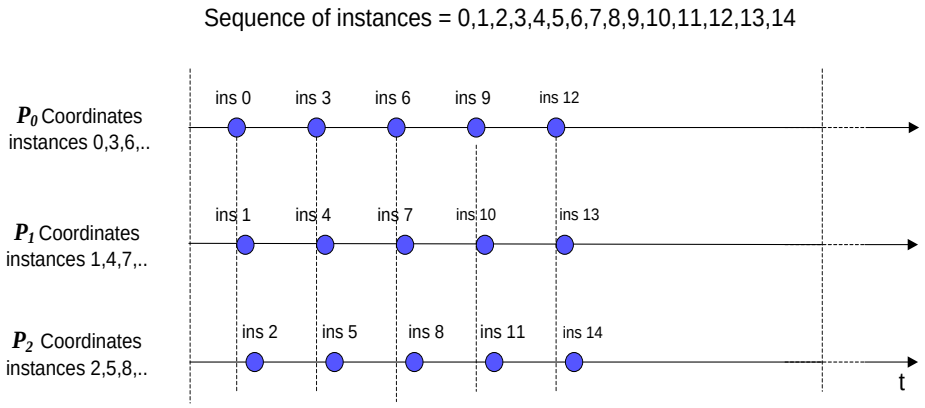

Figure 2.17: Servers suggesting with identical speed

Mencius is a consensus protocol that executes concurrent instances of simple consensus, so every server will be able to work separately and suggest requests according to the speed of its clients, and as soon as it receives the response from the majority, then the outcome of this instance can be learned and committed by all correct servers, regardless of the speed of other servers. However; the problem we are concerned about is that, if the application service did not receive instance *i-1*, but has already received instance *k*, where $k \geq i$. then instance *k* cannot be committed till all instances $< k$ should be received. Performance of the system is influenced by the speed of servers. The application service will commit requests according to the slowest server, which substantially degrades performance.

Figure 2.18 reflects the situation when servers working with different speeds. Servers P_0 and P_1 have the same speed and faster than server P_2. Instances *0,1,2,3,4* will be committed in sequence without any delay, but instance 6, and 7 have to wait for *5*, and instance *9* and *10* also has to wait for *8*. The system given here will commit instances according to the speed of server P_2. The next rule will address this point.

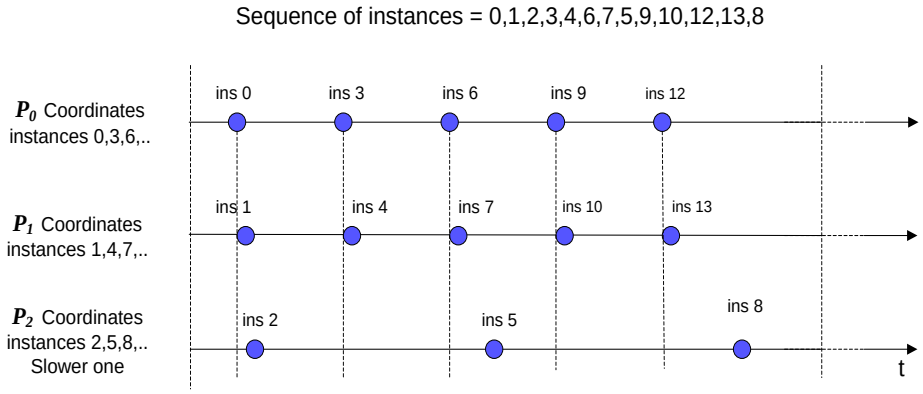

Figure 2.18: Servers suggesting with different speed

Rule2. When server *p* receives a *SUGGEST* for instance *i* and $i > Ip$, *p* updates *Ip* such that $Ip' = min\{k : p \text{ coordinates instance } k \wedge k > i\}$, before accepting the value and sending back an *ACCEPT*. *p* also executes skip action for all instances in range [*Ip*,

Ip') that *p* coordinates. The solution proposed here produces no outstanding messages by fast servers, because the slow ones will skip their turn. However the problem of crashed server is not solvable by this rule. We stated before that the underlying network connecting servers implements asynchronous FIFO communication channel using TCP as the underlying transport protocol, that implies if server *p* has sent SUGGEST for instance *i*, then by the time all correct servers receive instance *i*, either they have already received all instances < *i*, or they are slow, and they would skip their turn.

Figure 2.19 depicts how the index of instances is adapted to the speed of servers. Server P_2 after receiving instance *6* will skip *5* and the next instance that will be produced is *8*. Servers P_0 and P_1 will commit *6* and *7* without waiting for *5*, which was skipped. Next server P_2 after receiving instance *12* will skip *11* and the next instance that will be produced is *14*. Servers P_0 and P_1 will commit *12* and *13* without waiting for *11*, which was skipped as well. In this way the problem of different speeds is mitigated.

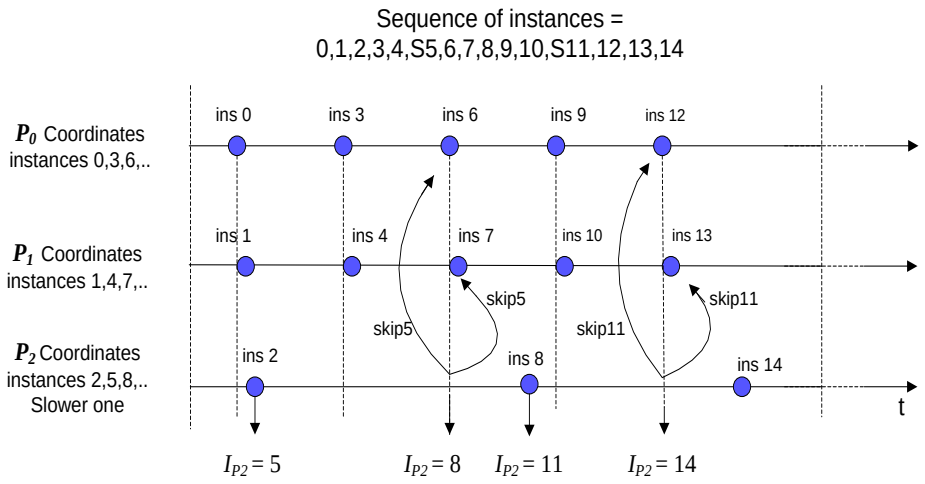

Figure 2.19: Servers applying rule2

Rule3. We assume that server *p* has been suspected by another server *q* to be failed, and let us consider that C_P is the smallest instance that was not learned by *q* and

should have been coordinated by *p*. Therefore, *p* will be revoked by *q* for all instances that *p* coordinates in the range $[C_P, I_q]$.

Figure 2.20 shows how server P_0 revokes P_2 for instances *(11,14,17,20)* that P_2 coordinates, bigger than instances *8* (the last instance received from P_2) and smaller than instance *21* produced by P_0, where $C_{P2} = 11$ and $Ip = 21$. so servers P_0 and P_1 will be able to commit all instances from *11* up to *19*. However; if server P_2 will continue crashed then one of servers P_0 or P_1 will raise again as the coordinator that will revoke P_2. Revocation will continue as long as P_2 crashed.

The number of times a crashed server is revoked affected by the range $[C_{P2}, I_{po}]$. If this range has one message then revocation needs to be carried on for each instance of simple consensus. Due to the use of all phases of the protocol we have more latency and message complexity, to reduce the extra cost generated by revocation we have to increase this range.

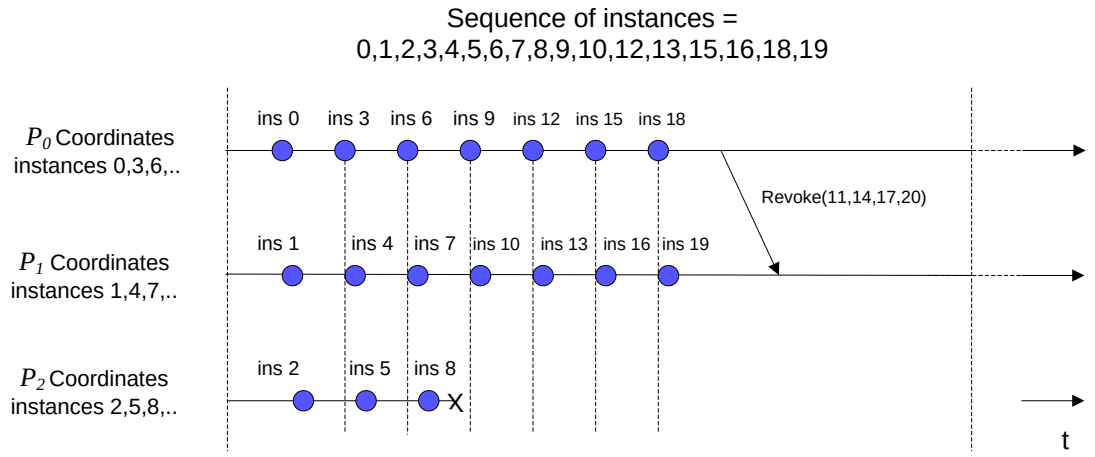

Figure 2.20: P_0 revokes P_2

Figure 2.21 exemplifies a system of 3 servers P_0, P_1, and P_2. P_2 learns instance 5 and sends it to P_0 only then crashes, so P_1 did not know about instance 5. P_0 raise up to revoke P_2 and starts to proposes on its behalf. In this example P_0 revokes P_2 for *8* by sending *PREPARE*, as P_1 does not know about *5* asks for it, P_0 sends learn 5 to P_1 and continues with revocation. Then P_0 *proposes no-op* for *8* after *ack* from the majority, after being *accepted* by the majority *learn* will be sent. This example is equivalent to

the one explained at section 2.5.2.6, *Total Order Protocol* with leader change, Figure 2.10.

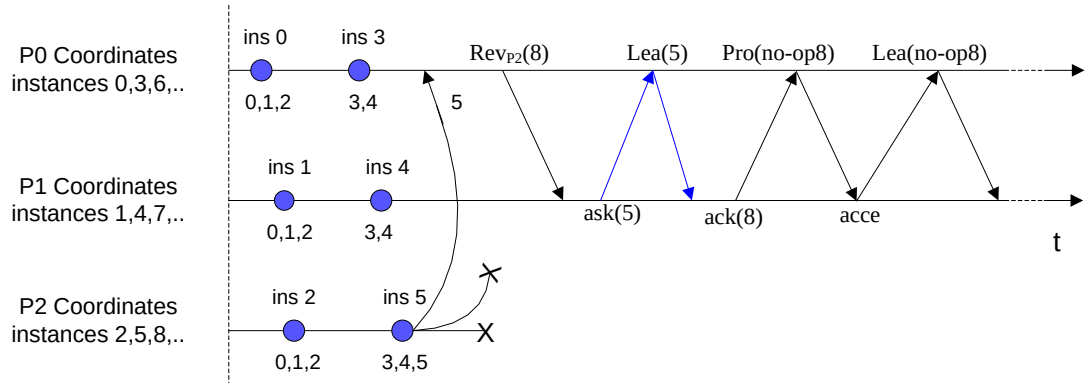

Figure 2.21: P_2 crashes after learning 5 and only makes P_0 learn

Any correct server suggesting *v* for instance *i*, a server *p* upon receiving *v* will update its *Ip* > *i*. According to rule 2 no outstanding messages, for all instances smaller than *i* that correct server *p* coordinate, it will either suggest *v* or skip and all correct servers will learn the outcome. However; according to rule 3 for all instance smaller than *i* that faulty server coordinate, will be eventually revoked by a correct server and other correct servers will learn the outcome. Thus, *v* for instance *i* will be committed. Unfortunately false suspicion, which is inherited from FD characteristics is not tackled in this point, but will be dealt with in the following rule.

Rule4. If a value $v \neq$ *no-op* was suggested by server *p*, but *p* learns that *no-op* was chosen for that instance, then *p* will suggest *v* again for a new instance *j* > *i*. If server *p* correct and not permanently suspected, it will succeed to suggest *v* again, server *p* will use a new index *Ip* according to Rule 2.

Figure 2.22 shows that P_2 was suspected by P_0 and revoked for instances *5, 8, 11* and *14*. P_2 did not succeed to finish instance *5*, but according rule 2 P_2 will learn that *no-op* has already been chosen for instances *5, 8, 11,* and *14* so it will try again using instance *17*.

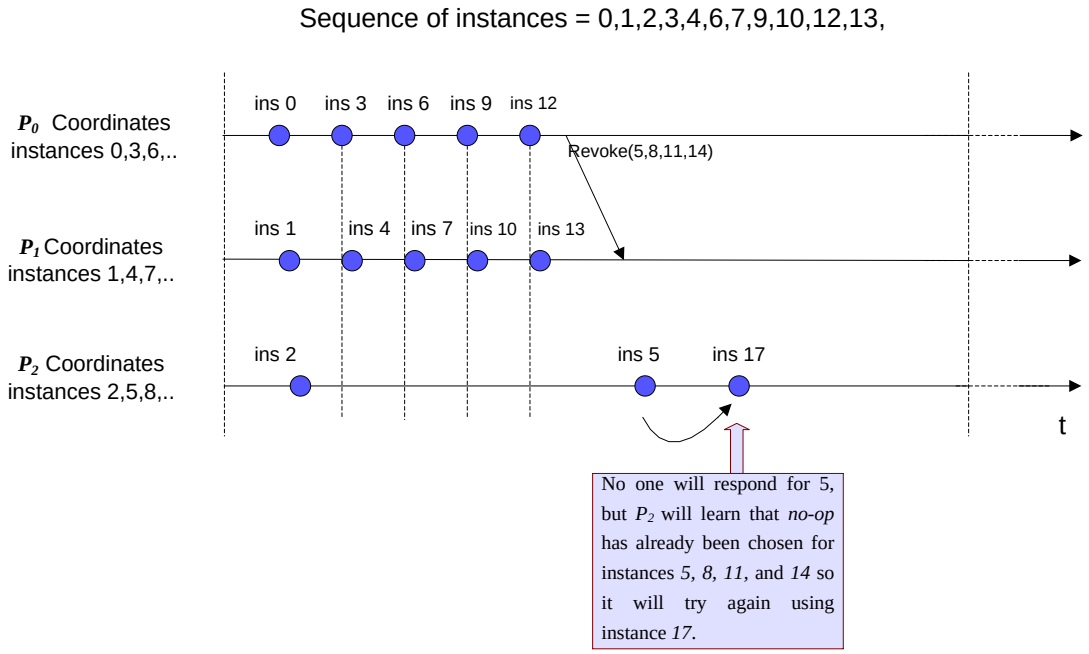

Figure 2.22: Suspected server tries again

Ω failure detector has been shown to be the weakest failure detector. This model of Ω guarantees that there is a time after which only one process is correct and not suspected by the other correct processes, but some correct processes may be permanently suspected. A stronger failure detector must be used to eliminate that permanently false suspicion, which is called $\Diamond P$. By the definition of $\Diamond P$, there exist a time *t* after which process *p* will not be suspected. The need for a stronger failure detector is necessary because leader change in Paxos might result of permanently false suspicion. The newly elected leader will fully replace the old leader. Nevertheless, this is not the case with Mencius; leader change in Mencius should be temporary, which takes place at revocation time only. If a leader was falsely suspected by failure detector and revoked then this leader must be able to come back as a leader and removed false suspicion.

Following these four rules the protocol is correct, but in order to increase efficiency some optimizations must be introduced.

Optimization 1

No *SKIP* message will be explicitly sent separately from server *q* to server *p*. Alternatively, server *q* sends *ACCEPT* as a response for *SUGGEST* received from *p*,

implies that future suggest message sent by *q* to *p* for any client request will have instance number higher than *i*.

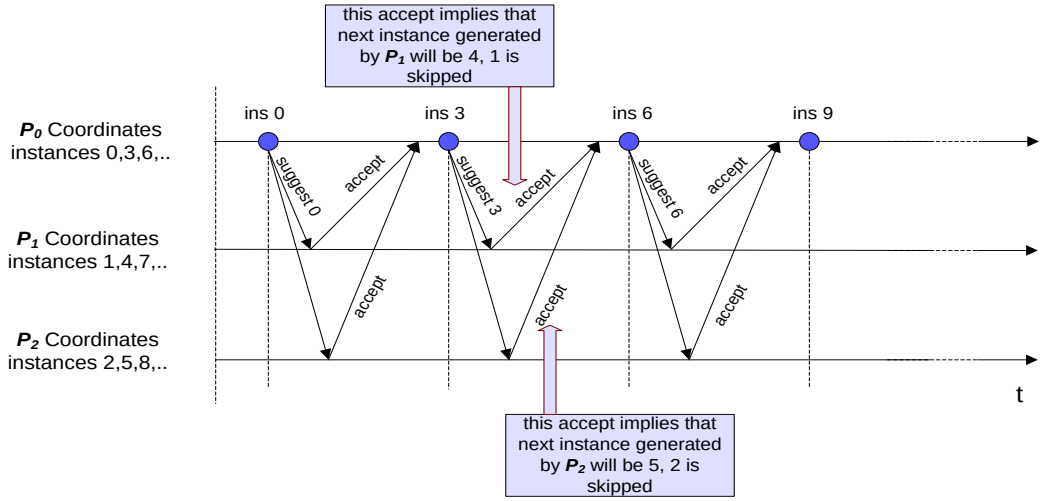

Figure 2.23: No need for skip message

This mechanism can also be applied between server *q* and other server *r*. Server *q* piggyback *SKIP* messages on future *SUGGEST* instead of *ACCEPT*.

Figure 2.23 reflects the behaviour of both servers P_1 and P_2 according to optimization 1. Server P_1 by sending *ACCEPT* as a response for *SUGGEST 3* will promise P_0 that next instance coordinated by P_1 will be higher than *3*, which is *4*.

The same applies for P_2, which will promise P_0 that next instance coordinated by P_2 will be higher than *3*, which is *5*. This optimization eliminates the need for *SKIP* message.

Optimization 2

No SKIP message will be sent immediately between server *q* and *r*. Alternatively, *q* waits for future SUGGEST from *r*, indicating that future suggest message for any client request will have instance number higher than *i*. Optimization 1 and 2 implies that no SKIP message that will be sent explicitly**.** Optimization 2 is different of optimization 1 and it is needed in case of two idle servers.

The implementation of optimization 2 creates a problem between two idle servers. Figure 2.24 shows that in case we have only one server P_0 suggesting values and the other two P_1 and P_2 are idle, consequently P_0 will learn *SKIP* message propagated to him from both according to optimization 1, and it will never be blocked, as there is no communication going on between server P_1 and P_2, they will learn nothing from each other, which will create a gap, and as a result of that they will be blocked as long as they are idle. We use a simple accelerator rule to limit the number of outstanding SKIP messages before P_1 and P_2 start to catch up.

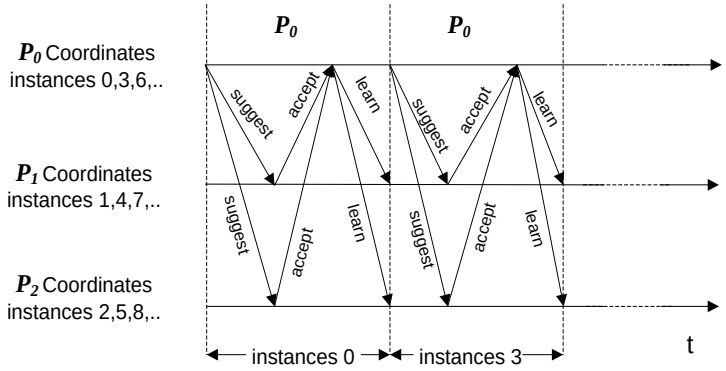

Figure 2.24: Sequence instances from P_0 only

Table 2.2 can be explained as following:

Row 1: 3 servers P_0, P_1 and P_2. P_0 suggesting values and the other two P_1 and P_2 are idle.

Row 2: P_0 suggests value for v_0, which is learned by P_1 and P_2

Row 5: P_0 suggests value for v_3, P_1 learns it and skips 1, which will be learned by P_0 only, and P_2 learns it as well and skips 2, which will be learned by P_0 only.

Row 3: P_0 and P_1 learn skip 1, but not P_2

Row 4: P_0 and P_2 learn skip 2, but not P_0

This explains how the problem between two idle servers builds up.

1	P_0	P_1	P_2
2	0	0	0
3	1 skip	1 skip	X
4	2 skip	X	2 skip
5	3	3	3
6	4 skip	4 skip	X
7	5 skip	X	5 skip

Table 2.2: idle servers problem

Accelerator 1

The propagation of *SKIP* message between two idle servers occurs when the total number of outstanding *SKIP* messages is larger than some constant α, or the messages have been postponed for more than some time τ. The use of *SKIP* message will have negligible side effect on the message complexity of *Mencius*, as it is used infrequently, message complexity for *Mencius* is *3(n-1).*

Optimization 3

We assume that server p has suspected server q to be failed, and let us consider that C_{P0} is the smallest instance that was not learned by p and has been coordinated by q. For some constant β, q will be revoked by p for all instances that q coordinates in the range $[C_q, I_P + 2\beta]$ *if* $C_{P0} < I_P + \beta$. β represents the number of instances that will be revoked in advance by server p , those instances will be greater than I_P. Comparing the implementation of rule 3 with optimization 3 shows that, optimization 3 will reduce the number of times needed to revoke a suspected server.

2.7.1 Choosing parameters

Accelerator 1 and optimization 3 requires the use of three parameters τ, α, *and* β. The value of these parameters should be engineered very carefully.

The value of τ should be large enough, which makes the cost of *SKIP* messages acceptable. But, a larger τ, results in a commit delay for requests received by idle P_1 and P_2 from P_0. Luckily, when idle, the clients connected to P_1 *and* P_2 do not send

any requests, so from a client's point of view the extra delay has little impact. 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 τ = 50 ms, Accelerator 1 generates at most 20 *SKIP* messages per second, if α is large enough. The CPU processing power and network bandwidth needed to process and transmit these messages are negligible.

(2) The extra delay of at most 50 ms added to the propagation of the *SKIP* messages, which could be attributed to packet loss or network delivery variance.
If τ has been chosen large enough, then constant α is used to limit the number of outstanding *SKIP* messages before P_1 and P_2 start to catch up. To reduce the overhead of sending α *SKIP* messages by factor of α, then all α *SKIP* messages are combined into one *SKIP* message. For example, if we set α to 100, this reduces the cost of *SKIP* message by 99%.
β defines an interval of instances: if a server q is crashed and Ip is the index of a non-faulty server p, then in steady state all instances coordinated by q and in the range $[Ip, Ip + k]$ for some $k : \beta \leq k \leq 2\beta$ are revoked. In order to reduce the impact of servers' inactivity we choose a large β to guarantee that while crashed other servers will not slow down, this also reduces the number of times when the crashed server will be revoked.

2.7.2 Revocation in *Mencius*

In *Mencius* revocation will have the following assumption:

1. More than one revocation can take place at the same time for a different number of servers.
2. Any server can be revoked by one server only [LAM01, MJM08].

In this section we will explain revocation in *Mencius*. Revocation is an operation carried by some correct servers to coordinate on behalf of the suspected server. This means that all instances that should have been coordinated by the suspected server are replaced with *no_op*. This operation helps correct servers to commit all outstanding

messages that have already been learned. We have to stress that a learned instance *i* will not be committed until all instances less than *i* are learned and committed.

Figure 2.25(a) shows revocation in a system that consists of three servers. P_0 suspects P_2 and starts revocation from the smallest instance *2* that is not learned by P_0 and should have been coordinated by P_2. In order for correct servers to commit values learned for instances larger than 2, P_0 will revoke all instances in range *[2, 9+2*5)* that should be coordinated by P_2 (the range is calculated according to Rule 4 and

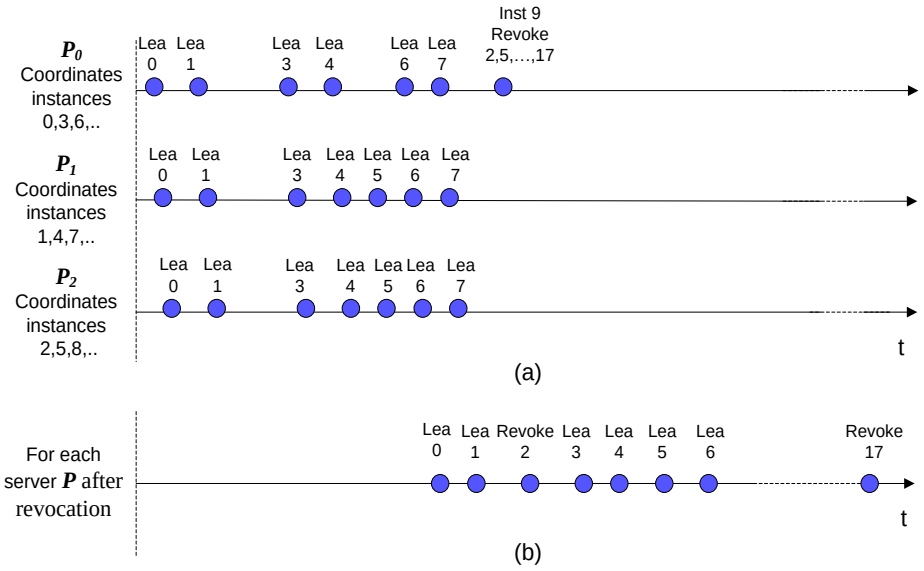

Figure 2.25: (a) P_0 revokes P_2 using Instance 9, (b) the outcome of revocation is learning instance 2 by P_0

Optimization 3 assuming that $\beta = 5$), P_0 will revoke instance *2, 5,* up to *17* using its own instance *9* as a round number, and 3 phase protocol. Revocation is carried out instance by instance, so P_0 will start revoking instance 2 using 9 a round number, then after finishing this instance will increase its round number to 12 to revoke instance 3, and so on up to instance 17.

The outcome of each instance depends on the state of the majority that will participate in that instance. If all of this majority report to P_0 in phase 1 that no value was chosen, then, in phase 2, P_0 will propose *no_op* for that instance, this is the case with revocation of instance *2*. But in case of revocation instance *5*, in phase 1 some

server reported to P_0 that it has already accepted or learned this instance, so the outcome was learning *5*. This process continues up to instance *17*

2.8 Summary

We started this chapter by exploring the three approaches designed to circumvent the impossibility result [FLP85]. The following a few lines we will try to give a brief summary about each one of them:

- Fail Signal, this approach has an advantage of perfect *Failure Detector* that makes no mistake. Nevertheless, it has higher cost regarding the number of physical machines (each node must have at least 2 machines) and also has higher message complexity. There are two factors that increase message complexity: (1) leader usually waits to receive responses from all correct FS nodes (not the majority), (2) there is redundancy in message generation by FS node constitutes. For example if FS node consists of 2 machines, then this FS node will generate two messages from each type. The redundant message will be considered as duplicate and discarded by the recipient.

- Randomized approach, it is a decentralized protocol that has no leader. The main disadvantage of this protocol is that the number of rounds and messages needed to arrive to a decision is unknown.

- Deterministic approach, in this approach we explored two protocols. We investigated Chandra and Toueg protocol [CT91, CT96], and Paxos [LAM01, LAMO6]. These are the two well-known asynchronous deterministic protocols designed to circumvent FLP impossibility result [FLP85].

We used a study that analyzes their performance to help us deciding which one of them will be the most suitable to use as underlying protocol for $[Mencius]^N$. This study is carried out in [UHS+04] which titled “Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm”. In this study, they arrived to the following conclusion “We evaluate the steady state latency in (1) runs with neither crashes nor suspicions, (2) runs with crashes and (3) runs with no crashes in which

correct processes are wrongly suspected to have crashed, as well as the transient latency after (4) one crash and (5) multiple correlated crashes.

The results. Our main finding is that, although the two algorithms have comparable performance in scenarios (1), (2) and (4), *the Paxos algorithm performs significantly better in scenarios 3 and 5*". This clearly shows why Paxos was chosen as underlying protocol in *Mencius* and *[Mencius]*^{N}.

Chapter 3

Performance Assessment of *Mencius*

3.1 Introduction

Having studied and analyzed *Mencius* thoroughly in the previous chapter, we here expand on certain concerns about *Mencius* performance. We decided in this chapter to assess *Mencius* regarding its performance in relation to the assumption of the occurrences of false suspicion. In *Mencius,* they assume that false suspicions rarely occur in practice. In contrast to Mencius, we consider in this chapter that false suspicions can occur frequently. The result of the new consideration will be reflected in an increase of revocation overhead in the system. The problem is two fold; first is revocation overhead, and second is the frequency of false suspicion occurrence. The latter will be dealt with in chapter 4, while the first will be dealt with in this chapter.

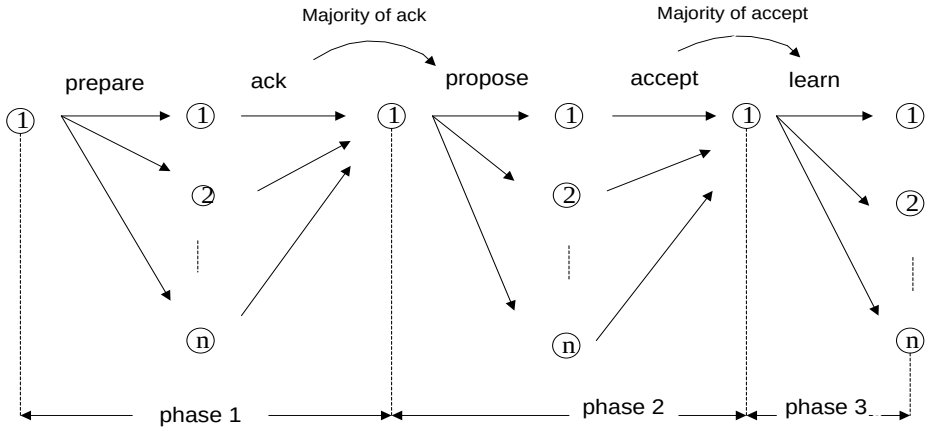

Figure 3.1: *Mencius* with three phases

As described in the previous chapter, Paxos can be implemented using either 2-phase (Classic Paxos) or 3-phase. Even though the latter has more phases than the former,

but message complexity in 2-phase version is $\{3(n-1)+n^2\}$ which is higher than the 3-phase $\{5(n-1)\}$. Because of that advantage of message complexity *Mencius* implementation uses 3-phase version of Paxos. The protocol will go through all those phases when there is a suspicion of failure or crash, however; when there is no suspicion of failure or crash, the protocol will execute the last 2 two phases only. Figure 3.1 gives an illustration about the 3-phase and their distinct message names. In the following sections more details will be presented about the function of each message.

3.2 Criticism of revocation in Mencius

In our analysis of revocation we will assume that P_i suspects P_j

Very important notations:

- I_{Pi}: is the index or sequence number of server P_i for the next simple consensus instance.
- C_{Pj} : is the smallest instance that was not learned by P_i and should have been coordinated by P_j
- β : defines the interval of instances that should be revoked in advance

In this section we will analyze revocation as implemented in *Mencius*, trying to highlight its drawbacks and side effects on the whole system. All instances in the range $[C_{Pj}, I_{Pi} + 2\beta]$ that should have been coordinated by the suspected server are revoked one by one by the revoker using the full protocol (3 phases), and also it has to propose its own requests if there is any (using 2 phases) as well, so the revoker will be over loaded during that period and slows down. In brief, as the revoker has to do two jobs at the same time, there is an increase in the demand for more CPU processing power. Revocation will produce unbalanced CPU utilization; the CPU of the revoker will be fully utilized, while the other correct servers are not. In higher rates of requests, the system suffers higher latency and lower threshold of saturation.

Revocation overhead is now analysed as shown in figure 3.2, in which we assume P_2 crashed and P_0 revokes it. Figures used in this example are quoted from the experiment carried out in [MJM08]: constant $\beta = 100{,}000$ which represents the number of instances that should be revoked in advance, and *FD* timeout is set to *5 seconds* starting from the moment the *TCP* connection is detected to be lost. *Mencius*

was capable of executing *3000* operation per sec (ops) for each server using 2 phase protocol. From all those figures we can calculate the total number of instances that should be coordinated by the revoker on behalf of the suspect server using 3 phase protocol *[(3000 * 5sec) + (2 * 100,000)/3servers) ≈ 81,666 op],* as the revoker uses 3 phases will be able to produce less than 3000 ops. In addition to that the revoker must coordinate its own requests if it has any using 2 phase protocol *[(2 * 100,000)/3servers) ≈ 66,666 op].* This illustrates the overhead that should be done by the revoker in case revocation is carried out one by one; the revocation method was described in chapter 2 section 2.7.3.

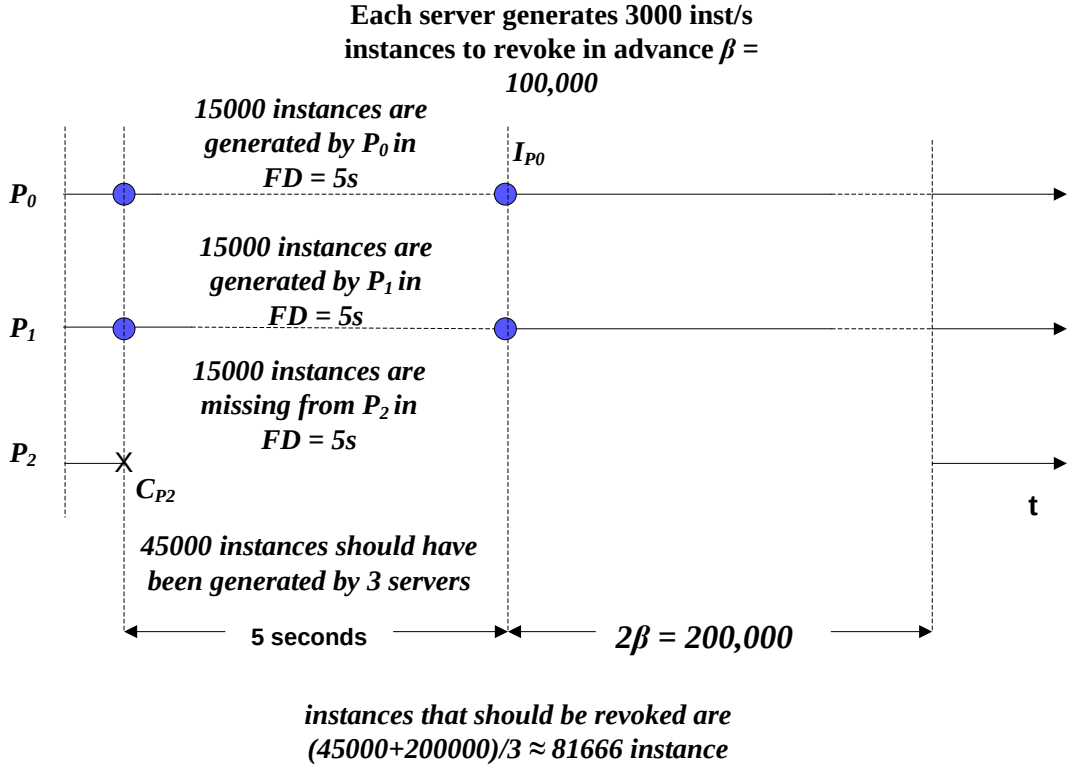

Figure 3.2: revocation overhead

Before presenting our solution, we will mention some facts about *Mencius* which will help in understanding our solution. FIFO implies that if server *q* coordinates instance *j* and *i*, where *j* < *i*, then all correct servers accept and learn *j* before *i*, and instances will be committed in sequence as well. So learning a value by itself does not mean that the value will be committed immediately, however; the system must

guarantee that those values will be committed in the right order, which might lead to some delay in committing instances. In case the system did not learn instance *i-1*, but has already learned instance *k*, where $k \geq i$. then instance *k* cannot be committed till all instances < *k* are learnt.

Mencius distinguishes two types of messages *(suggest = request and skip = no_op)* which can be proposed using 2-phase protocol and 3-phase protocol as well. In this chapter we show our concern regarding revocation in *Mencius.* During revocation process some server raises up to revoke a suspected server, the revoker tries to revoke a whole range of instances that should have been coordinated by the suspected server. Our solution says that the moment the new coordinator succeed in getting the majority of *acceptors* on *no-op* value then the whole range will be implicitly revoked as well. So there is no need to go in revocation one by one as the majority will promise not to accept any instance that could be proposed by the original coordinator in this range. The new modified version of *Mencius* addresses the problem of the high probability of crash and false suspicion that might be triggered more frequently by the FD.

3.3 Assumption and principles

Rule1. Each server *p* maintains its index number *Ip*, for a system that has three servers P_0, P_1, and P_2 then $I_{P0} = 0,3,6,\ldots$; $I_{P1} = 1,4,7,\ldots$; and $I_{P2} = 2,5,8,\ldots$, a server *p* suggests the client's request to consensus instance *Ip* and updates *Ip* to the next instance. If the speed of suggesting values of all servers is at the same rate, then rule 1 is sufficient for good performance. Figure 2.17 reflects the ideal system, all servers working with the same speed.

Rule2. When server *p* receives a *SUGGEST* from *q* for instance *i* and *i > Ip*, *p* updates *Ip* such that $Ip' = min\{k : p \text{ coordinates instance } k \wedge k > i\}$, before accepting the value and sending back an *ACCEPT*. *p* also executes skip action for all instances in range *[Ip, Ip']* that *p* coordinates. The solution proposed here produces no outstanding messages by fast servers, because the slow ones will skip their turn. However the problem of crashed server is not solvable by this rule.

Rule 3- We assume that P_i suspects that P_j has failed, and let C_{Pj} be the smallest instance that should have been coordinated by P_j and not learned by P_i. P_i revokes P_j for all instances in the range $[C_{Pj}, I_{pi})$ that P_j coordinate. Revocation is carried out for instance C_{Pj} only unlike *Mencius* (*Mencius* revokes all instance in that range one by one), there are only two possible outcomes of this revocation:

1- if phase 1 show no value was chosen then it will propose *no_op* in phase 2 for the whole range.
2- Otherwise, it proposes the possible consensus outcome by phase 2.

If case 1 was the outcome of revocation then all instances $k \geq C_{Pj}$ and $k < I_{pi}$ will be revoked automatically using only one Paxos instance and all of them will have *no-op* value.

Rule4. If a value $v \neq$ *no-op* was suggested by server *p* for some instance *i*, but *p* learns that *no-op* was chosen for instance *i*, then *p* will suggest *v* again for a new instance *j* > *i*. If server *p* is correct and not permanently suspected, it will succeed to suggest *v* again; server *p* will use a new index *Ip* according to Rule 2.

Optimization 1

This point will be explained according to the following system context; 3 servers *p, q,* and *r*. Server *p* is active while the other two servers *q* and *r* are idles.

No *SKIP* message will be explicitly sent separately from servers *q* and *r* to server *p*. Instead, servers *q* and *r* send *ACCEPT* as a response for *SUGGEST* received from *p* for instance *i*, implies that future *SUGGEST* message sent by *q* and *r* to *p* will have instance number higher than *i*.

In addition, this mechanism can also be applied between server *q* and the other server *r*. Server *q* piggybacks *SKIP* messages on any future *SUGGEST* sent to *r*.

Optimization 2

This point will be explained according to the following system context; 3 servers *p, q,* and *r*. Server *p* is active while the other two servers *q* and *r* are idle.

No SKIP message will be sent immediately between server *q* and *r*. Instead server *q* waits for future SUGGEST from *r* for instance *i*, indicating that future *SUGGEST*

message generated by *q* will have instance number higher than *i*. Optimization 1 and 2 implies that no *SKIP* message that will be sent explicitly**.**

Accelerator 1

The propagation of *SKIP* message between two idle servers occurs when the total number of outstanding *SKIP* messages is larger than some constant α, or the messages have been postponed for more than some time τ. The use of *SKIP* message will have negligible side effect on the message complexity of *Mencius*, as it is used infrequently, message complexity for *Mencius* is *3(n-1).*

Optimization 3

We assume that server *q* has suspected server *p* to be failed, and let us consider that C_{P0} is the smallest instance that was not learned by *p* and has been coordinated by *q*. For some constant β, *q* will be revoked by *p* for all instances that *q* coordinates in the range $[C_q, I_P + 2\beta]$ *if* $C_{P0} < I_P + \beta$. β represents the number of instances that will be revoked in advance by server *p,* those instances will be greater than I_P. Comparing to the implementation of rule 3 with optimization 3 shows that, optimization 3 will reduce the number of times needed to revoke a suspected server.

3.4 Protocol description

Our multi ordering protocol as a modified version of *Mencius* and will be explained for three different cases. Every case will give a thorough explanation and will elaborate on different aspects of the modified protocol. As the operational environment changes with passage of time, as in t_1 the state of the system is different from that in t_2, and in order to cover all states we will put some assumption that should be followed in every case. We may assume that in t_1 the system has no crash and no false suspicion but there is no guaranty that this will continue for ever, so the system in t_2 may have different operational conditions. First we will start by applying the strongest assumptions about the system, and then we will remove those assumptions gradually to get to the weakest one.

Case 1- Assumptions:

1. We assume that all servers working correctly according to their specification, and that no crash and no false suspicion occur that is, *FD* makes no mistakes.
2. All servers propose requests in the same speed.

Each server coordinates its own requests using a Paxos instance for each request applying its own index numbers I_{pi} as a consensus number, server P_i suggests the received request using its current I_{pi}, and updates I_{pi} to indicate the next instance it will coordinate, so every time a unique index number is used which will not collide with other servers index numbers as shown in figure 3.3(a). The outcome of each Paxos instance will be handed to all servers as learned message, individual servers receives learned message from different servers and commit them in a sequential order as shown in figure 3.3(b).

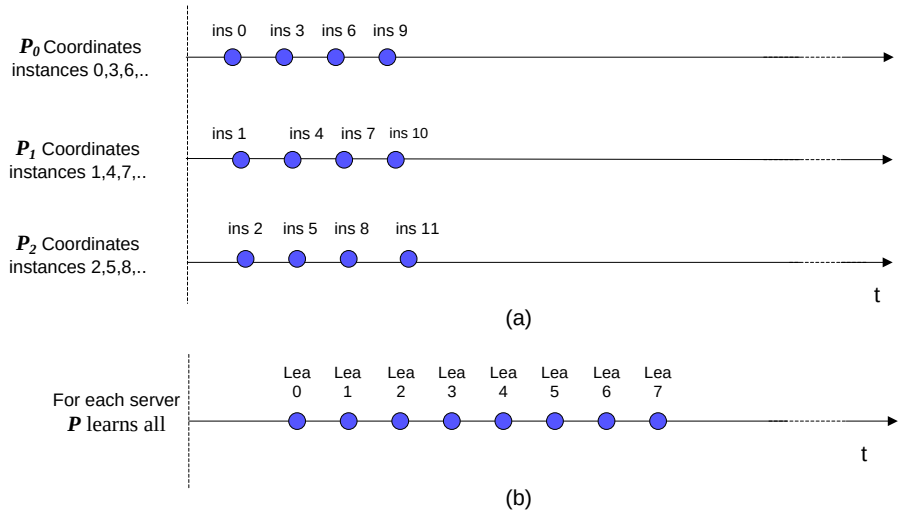

Figure 3.3: (a) each server runs its own Paxos instances,
(b) each server learns and commits all instances in sequence

Case 2- Assumptions:

1. We assume that all servers working correctly and no false suspicion as well (*FD* makes no mistakes).
2. Servers work with different speeds.

The speed of each server depends on client's requests, so when the client needs no service, then no requests will be handed to their server, figure 3.4(a) shows that server P_2 has nothing to coordinate which explains the gap shown in figure 3.4(b). All three servers will learn instance 0,1, ,3,4, ,6,7,…., but not 2, 5 and so on, they will be able to commit instance 0, and 1 only, but instance 3, and 4 will be blocked till server P_2 coordinate instance 2, and instance 6, and 7 will be blocked till server P_2 coordinates instance 5 as well. In order to prevent the system of being blocked for a long period of time we resort to *SKIP* message with a *no_op* value sent by the slow server to speed up the system, as was explained at Optimization 1. According to optimization 1 when P_2 receives suggest from P_0 for instance 3 or suggest from P_1 for instance 4 will respond by sending *accept* which will imply that P_2 is skipping 2,

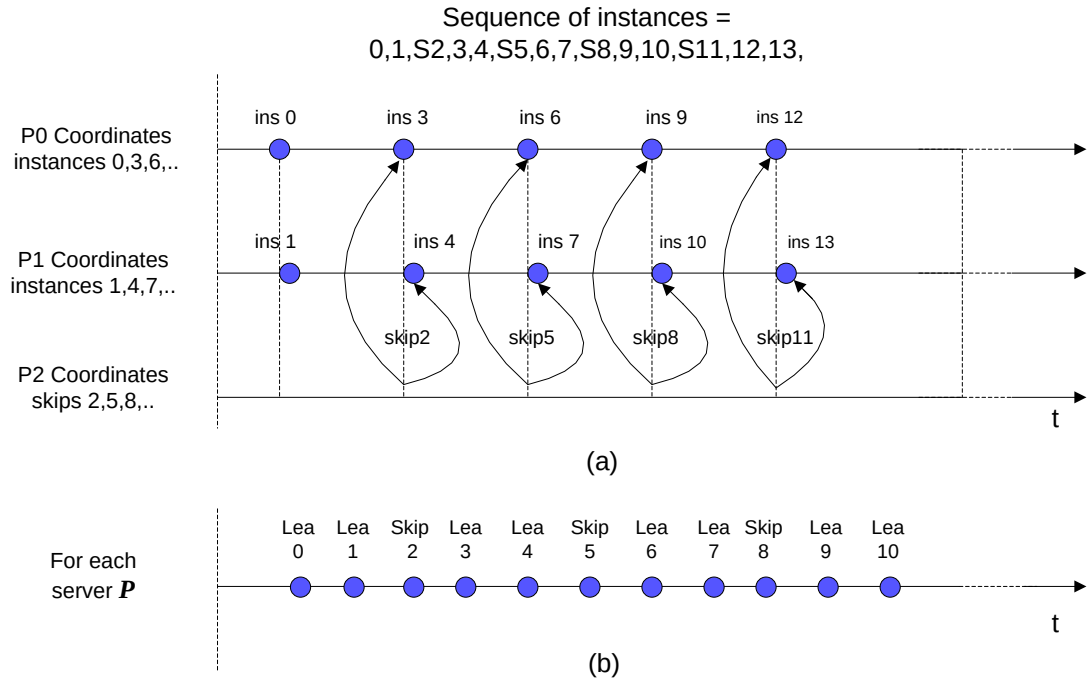

Figure 3.4: (a) server P2 has got no requests to coordinate,
(b) each server learns all instances and commit them

Figure 3.5(a) shows how accelerator is used to remove any outstanding messages between two idle servers. We set $\alpha = 4$, and, for simplicity, we are not using any timers.

1- After receiving instance *3* from P_0, P_1 sends skip *1* to P_0 and starts counting outstanding messages towards idle P_2. After receiving instance *3* from P_0, P_2 sends skip *2* to P_0 and starts counting outstanding messages towards idle P_1.
2- After receiving instance *6* from P_0, P_1 sends skip *4* to P_0 and continues counting outstanding messages towards idle P_2. After receiving instance *6* from P_0, P_2 sends skip *5* to P_0 and continues counting outstanding messages towards idle P_1.

3- After receiving instance *9* from P_0, P_1 sends skip *7* to P_0 and continues counting outstanding messages towards idle P_2. After receiving instance *6* from P_0, P_2 sends skip *8* to P_0 and continues counting outstanding messages towards idle P_1.

4- This is going to be the last one as we set $\alpha = 4$. After receiving instance *12* from P_0, P_1 sends skip *10* to P_0 and continues counting outstanding messages towards idle P_2. After receiving instance *12* from P_0, P_2 sends skip *11* to P_0 and continues counting outstanding messages towards idle P_1.

5- Accelerators of both idle servers P_1 and P_2 will be triggered figure 3.5e. P_2 will send *skip(2,5,8,11) to* P_1, and P_1 will send *skip(1,4,7,10) to* P_2. this move will remove any outstanding messages for both of them, and they will be able to commit everything.

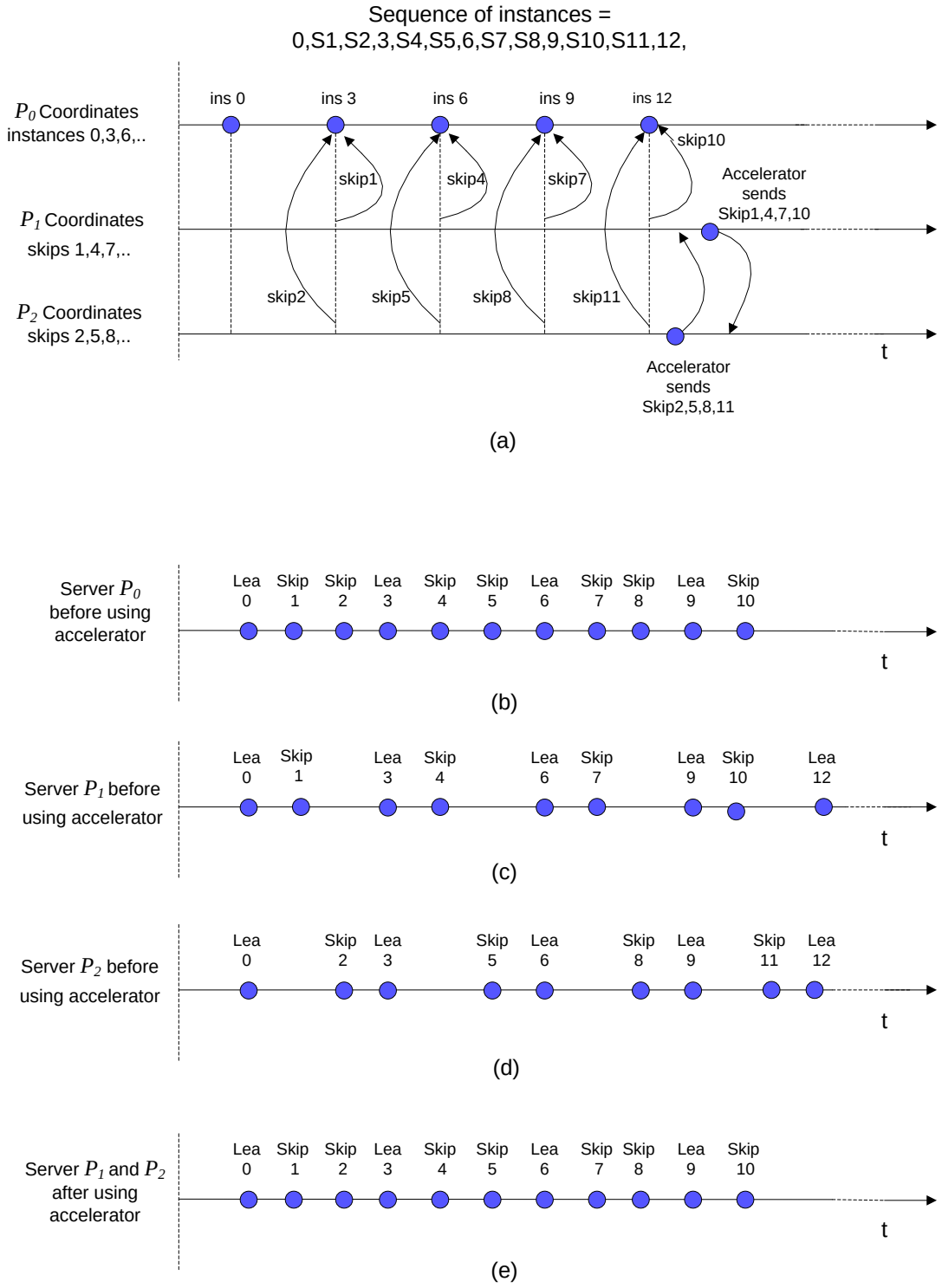

Figure 3.5: (a) servers *P_1 and P_2* idles, (b) server *P_0* learns everything instantly (c) *P_1* learns from *P_0* and its own skips only, (d) *P_2* learns from *P_0* and its own skips only, (e) accelerators are triggered and both *P_1* and *P_2* learns and commits everything

Case 3- Assumptions:

1. We assume that some server can crash or could be falsely suspected (*FD* makes mistakes).
2. Servers work with different speeds.

Revocation is a process carried out by some server on behalf of the suspected one. This means that all instances that should have been coordinated by the suspected server are replaced with *no_op*. This operation helps correct servers to commit all outstanding messages that have already been learnt. We have to stress that a learned instance *i* will not be committed until all instances less than *i* are learned and committed. The system state will only change the moment a message is committed, but not learned only. This fact was the foundation on which we build our revocation system.

Figure 3.6(a) shows revocation in a system that consists of three servers. P_0 suspects P_2 and starts revocation from the smallest instance *2* that is not learned by P_0 and should have been coordinated by P_2. In order for correct servers to commit learned values for instances larger than *2*, P_0 will revoke all instances in range

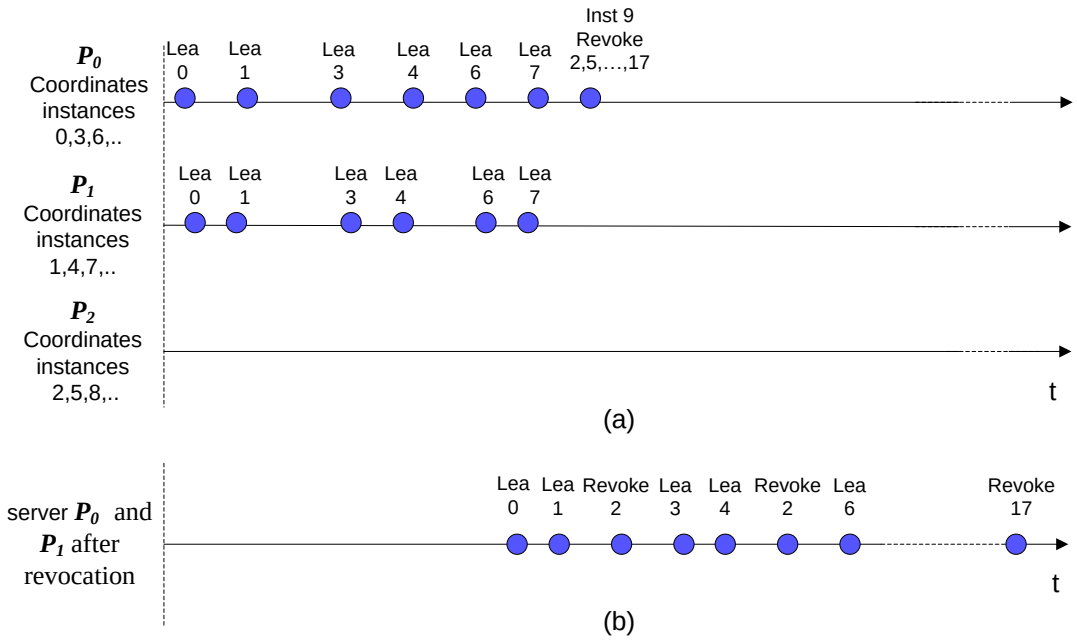

Figure 3.6: (a) P_0 revokes P_2 using its own Instance 9, (b) after revocation each correct server will be able to commit any outstanding messages.

[C_{p2}, I_{P0} + 2β] that should have been coordinated by P_2 (the range is calculated according to Rule 4 and Optimization 1 assuming that $\beta = 5$, $C_{p2} = 2$, $I_{P0} = 9$), so the range will be *[2, 9+2*5)*. P_0 will revoke P_2*'s* instances *2, 5,* up to *17* using its own instance *9* as a round number and 3-phase protocol. The outcome of this depends on the state of the majority that will participate in this instance. If all of this majority report to P_0 in phase 1 that no value was proposed or learnt, then, in phase 2, P_0 will succeed in revoking the whole range as shown in figure 3.6(b) using *9* as a round number, following this revocation all outstanding message will be committed.

In some circumstances as depicted in figure 3.7(a) some servers (for example P_1*)* learnt instance *5* and *2* from P_2, however; P_0 did not receive anything from P_2. In such situation P_0 tries to revoke P_2 using its own instance *9* as a round number. P_0 will revoke a range *[2, 9+2*5)* starting from instance *2*, which represents the smallest instance that was not learnt by P_0. As one of the majority that participated with P_2 for instance *5* (for example P_1*)* will *NACK (2,5)* in phase 1 to P_0, then, P_0 learns these two instances and aborts this round. Figure 3.7(b) shows that both P_0 and P_1 will be able to commit any outstanding values.

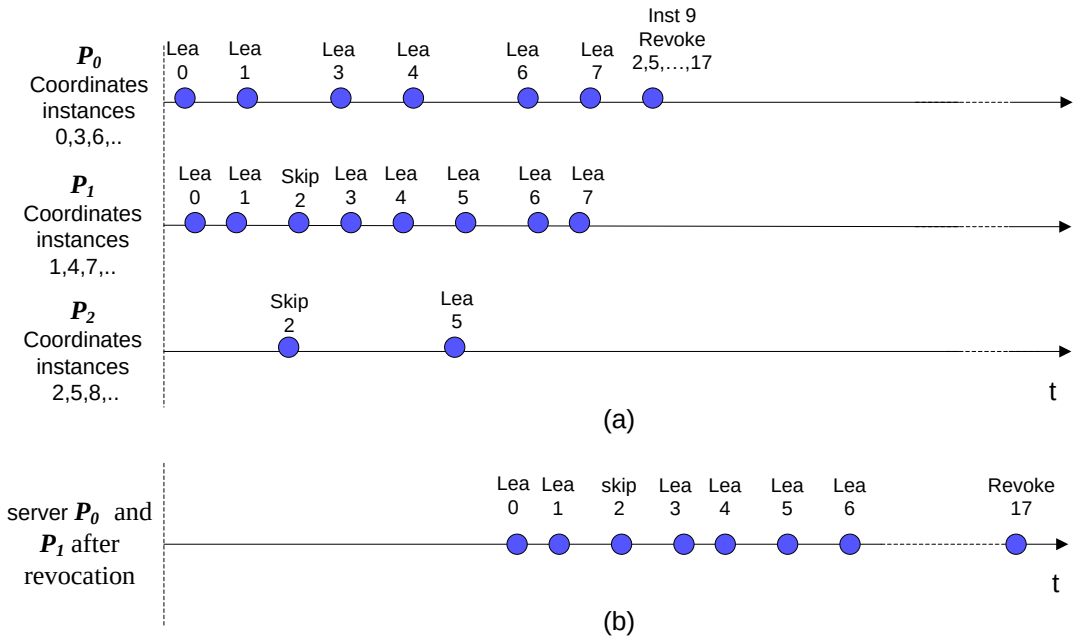

Figure 3.7: (a) P_0 revokes P_2 using its own Instance 9,

(b) P_0 learns 2 and 5, then aborts revocation

Figure 3.8 shows 5 server system (P_0, P_1, P_2, P_3, *and* P_4), only server P_0 is active while all other 4 servers are idle. P_0 learns everything from himself and from the other 4 correct servers. P_1, P_2, P_3, *and* P_4 each one of these servers will learn values produced by P_0 and also will learn its own implicit skip only. For example P_1 will learn *0, 5* and *10* from P_0 and its own skips *1* and *6*. Another example P_2 will learn *0, 5* and *10* from P_0 and its own skips *2* and *7*, and so on for the other two servers.

P_1 suspects P_4 and tries to revoke all P_4*'s* instances in the range *[4, 15+2*10),* that is, it revokes *4, 9 . . . 34* of P_4, where $\beta = 10$, $C_{p4} = 4$, and $I_{P0} = 15$. P_0 knows the state of all servers in the system, and also each one of the other four servers knows about its state and P_0*'s* state. We assume that P_2 and P_3 participate with P_1 in revoking P_4, P_1 sends *PREPARE* to them informing them that it is going to revoke P_4 from instance *4* up to *34*. As P_2 and P_3 know nothing about P_4 acknowledging P_1. In phase 2 P_1 will propose *no-op* for the whole range [4 . . . 34]. These 3 servers P_1, P_2, and P_3 will learn no-op for the whole range, but not P_0 and P_4, because they did not take part in that revocation. The system state about P_4 will be as following:

1. P_0 learns skip 4 and skip 9
2. P_1 learns skip 4 up to skip 34
3. P_2 learns skip 4 up to skip 34
4. P_3 learns skip 4 up to skip 34
5. P_4 learns skip 4 and skip 9

If P_4 attempts to propose any *SUGGEST* for instance 14, it will learn about the revocation from 4 up to 34. As it waits for a majority of *ACCEPT*. Any majority should include at least one of P_1, P_2, or P_3. Then P_4 will abort this round and tries again using round number higher than 34 which is 39. When P_4 succeeds in *SUGGESTing* 39 that implies, it has already skipped 14, 19, 24, and 29. That makes the system consistent, because all servers learnt *no-op* for that range.

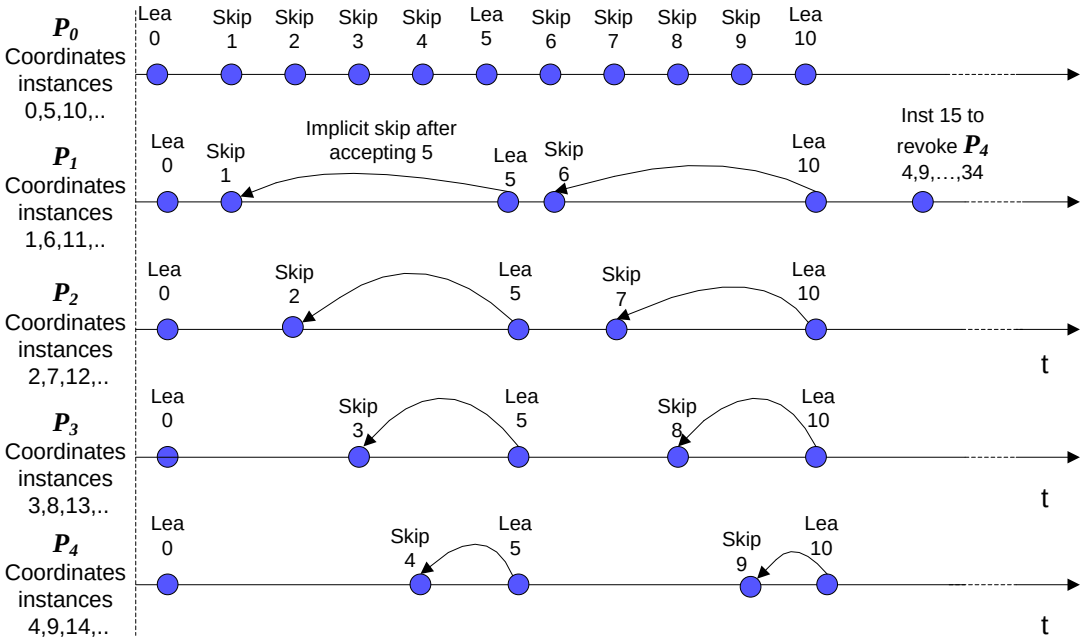

Figure 3.8: P_1 revokes P_4 using Instance 15

3.5 Summary

In this chapter we presented a revised version of *Mencius* built on a different fundamental assumption. This assumption says that false suspicion and crash occur frequently which is contrary to the *Mencius* assumption. Revocation overhead in *Mencius* is acceptable as long as false suspicion and crash occur rarely. We believe that if there is an increase in false suspicion and crash cases which occur more repeatedly, then it is reasonable to re-address the issues of revocation. As was explained in more detail through the course of this chapter, revocation can be achieved by using only one instance of Paxos to revoke a whole range.

The new mechanism of revocation reduces its overhead to the minimum. The revoker runs only one instance of Paxos to revoke the whole range of instances that should have been generated by the suspected or crashed server. When the revoker

finishes this instance goes back to its normal job. This will make the revoker and the other correct servers work with the same processing capacity.

Chapter 4

Protocol *[Mencius]*^{N}

4.1 Introduction

Mencius in failure-free situation outperforms Classic Paxos. By having multiple leaders, the throughput is increased under high client load and latency is lowered under low client load. However; the implementation of *Mencius* over wide-area network has several disadvantages and these disadvantages can substantially degrade *Mencius* performance. The wide-area network characterized with high latency, small bandwidth and the latency can have high variance. Using *Mencius* over wide-area network will be incurred with many problems such as false suspicion, latency, and bandwidth consumption of wide-area network. *Mencius* could have another problem which is node crash. Because each site in *Mencius* consists of one server, the crash of that server means all clients connected to it are blocked.

The following will be a brief illustration of the aforementioned problems. The first three points are related to the implementation of Mencius over wide-area network while the last one is related to the level of redundancy at site level (one server/site):

1. ***Latency:*** All instances in *Mencius* are executed using Paxos protocol, so each instance will be executed across wide-area network using 3 or 5 messages. The latency that incurs client requests depends on delays of wide-area network.
2. ***False Suspicion:*** The implementation of *Mencius* over wide-area network increases the chances of false suspicion occurrence in the system. Wide-area network as an asynchronous model characterized with unknown bounds on message delay and unpredictable workload. These attributes increase Fail Detector unreliability, which leads to an increase of its mistakes [FLP85]. Consequently, false suspicion results in poor performance as stated in rule 4 and optimization 3 [MJM08].
3. ***Bandwidth consumption:*** the problem of *Mencius* regarding this issue is that to finalize each request, *Mencius* will run an instance of Paxos either of 3 or 5 messages. The correct server will go through instances of Paxos that has 3 messages (propose, accept, and learn) when it is in a process of proposing its own

requests. While the revoker will go through instances of Paxos that has 5 messages (prepare, ack, propose, accept, and learn) when it is in a process of revocation. In our proposed work, both of theses case will be reduced to one message only.

4. ***Crash:*** Site crash in *Mencius* will block all clients connected to that site within its local-area network. In figure 4.1 If P_0 crashes, then any one of the correct servers P_1 or P_2 can replace it to fill in the gap by producing *no-op* messages. The crash is therefore partially solved, by allowing clients connected to the correct servers to make progress and therefore commit what they learn. However, requests issued by clients connected to the crashed server will be blocked as long as crash conditions exist. To compound the crash problem, scaling in the hardware integration process now increases reliability challenges [EZH08, BAU05] to modern systems making crash an inevitable problem.

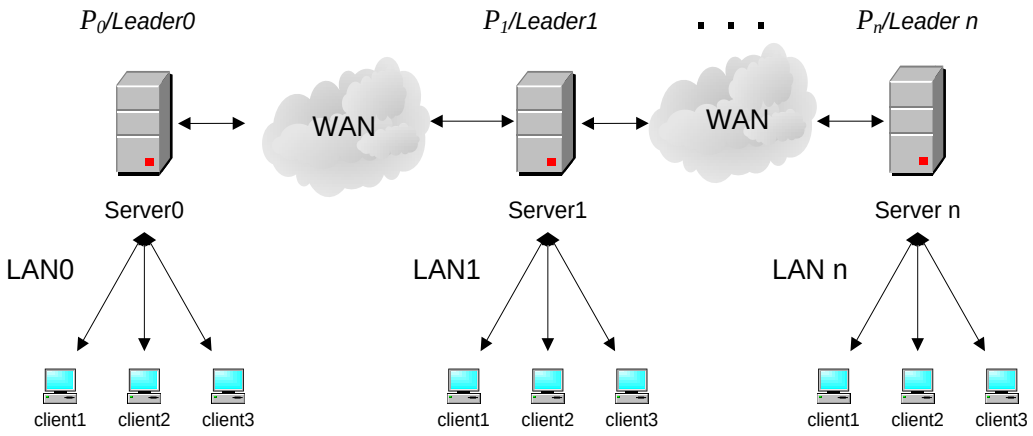

Figure 4.1: Network context of *Mencius*

This chapter will focus on the following problems: latency, unreliable Fail Detector, crash, and bandwidth consumption. Are there any ways in which these issues can be tackled? Our proposal is to build multiple-cooperative *Mencius* as a two-layer system. One layer consists of local *Mencius* and the second layer forms global *Mencius*. The underlying network connecting servers of each local *Mencius*

system will be based on asynchronous local-area network. However, the underlying network connecting global *Mencius* systems will be based on asynchronous wide-area network. Both of these systems exchange messages.

4.2 Two Level *Mencius*

Figure 4.2 represents a system that consists of two replicas or two levels: global replica and local replica. Global replica exists on a level of sites *N*, where $N \geq 2$. They communicate through an asynchronous network (WAN) to implement a replicated state machine. Each site (S^{SI}) represents one *Mencius* system, creating $[Mencius]^N$ systems.

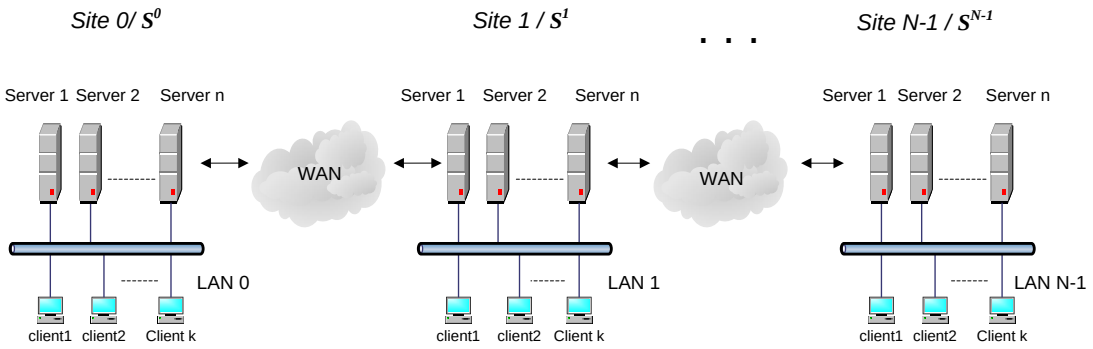

Figure 4.2: Network context

Local replica (site) on the level of servers, each local replica consists of *n* servers, forming a local *Mencius* system. Each site has also a number of clients which communicate with local servers. The underlying network connecting servers implements the FIFO communication channel, since TCP is used as the underlying transport protocol. This implies that messages between two correct servers are eventually delivered and delivered in order.

Each site should be implemented using $n \geq 2f+1$, where $f \geq 1$, replica processes that are fail-independent, hosted on distinct nodes (machines) connected using local area network. These replica processes are referred to as servers *(s).* At most *f* of these replicas can fail by crash at any time. Figure 4.3 details one of these sites. Servers

communicate through a fast and reliable network to facilitate the work of replicas and to increase the reliability of FD and reduce its mistakes

Sites are always found in working state as long as the majority are correct. Crash within each site is transparent and not exposed to the outside world. Working State means that the site is working correctly and is free of faults. All expected outputs are produced and each output is correct and sent to all relevant destinations. Site operates as per the specification of the program executed by that site, assuming each site at initialization is found in this working state. As each site forms a local *Mencius* system, server crash is dealt with in terms of revocation, which was described in detail in the previous chapter.

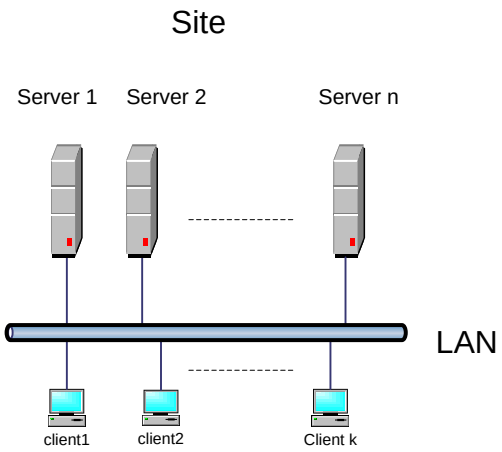

Figure 4.3: Site structure representing local *Mencius*

[Mencius]^{N} is a multi-leader state machine replication protocol that derives from *Mencius.* We consider each site as a leader that coordinates its own instances (figure 4.4.) Each site is equivalent to one *Mencius* that orders requests received from its own group of clients to generate its Local Commit Stream. This stream is then converted to Global Commit Stream on a global level.

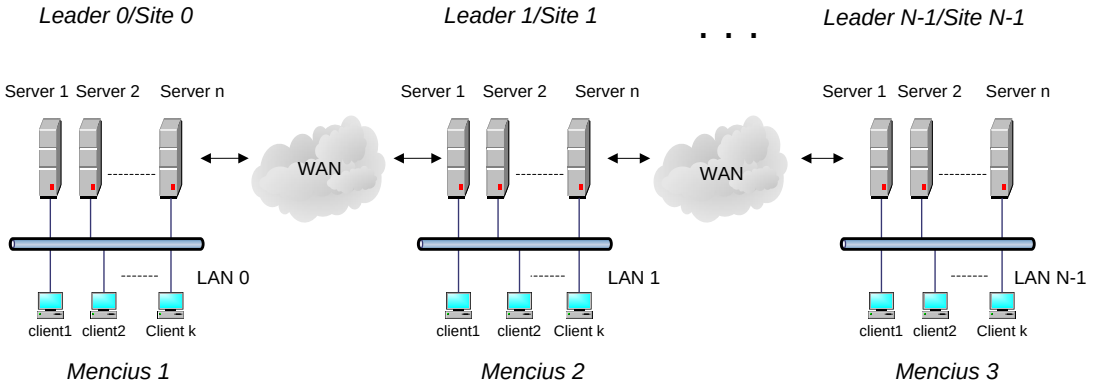

Figure 4.4: *[Mencius]*^{N} as a multi-leader protocol

4.3 Assumption

We make the following assumptions about *[Mencius]*^{N}:

- For each site, at most *f* servers can fail by crash at any time.
- Failure detector oracle: *[Mencius]*^{N} requires $\Diamond P$ [MJM08], a class of failure detectors that eventually guarantees all faulty servers and only faulty servers are suspected. The implementation of local replica of *Mencius* over local-area network facilitates its work and increases the reliability of its Fail Detector, which will result in decreasing FD mistakes.

4.4 Principles

In order to reduce redundancy, the implementation of *Mencius* within each site will adopt the same assumptions and principles mentioned in chapter 3. Every site is the coordinator of an unbounded number of instances (system of numbering instances is inherited from *Mencius*). For every site, there is a specific number of instances assigned to other sites between consecutive instances that site coordinates. Each site will produce its own local commit stream. That stream then will be transferred by each server to a Global Commit stream. Each site, S^{SI}, maintains its site index number, I_{SI}. The number of sites N could be any number ≥ 1, but for convenience in this work we assume a system that has three sites ($iN+SI$ to site S^{SI}, where $i \in N_0$ and

SI(site index) $\in \{0, \ldots, N-1\}$, *for* $N = 3$) S^0, S^1, and S^2 then $I_{S0} = 0,3,6,\ldots$; $I_{S1} = 1,4,7,\ldots$; *and* $I_{S2} = 2,5,8,\ldots$.

Figure 4.5 shows how local streams are formed then converted to a global one. It is a fact that all servers in a site will produce the same local stream and also will be able to map it to global stream.

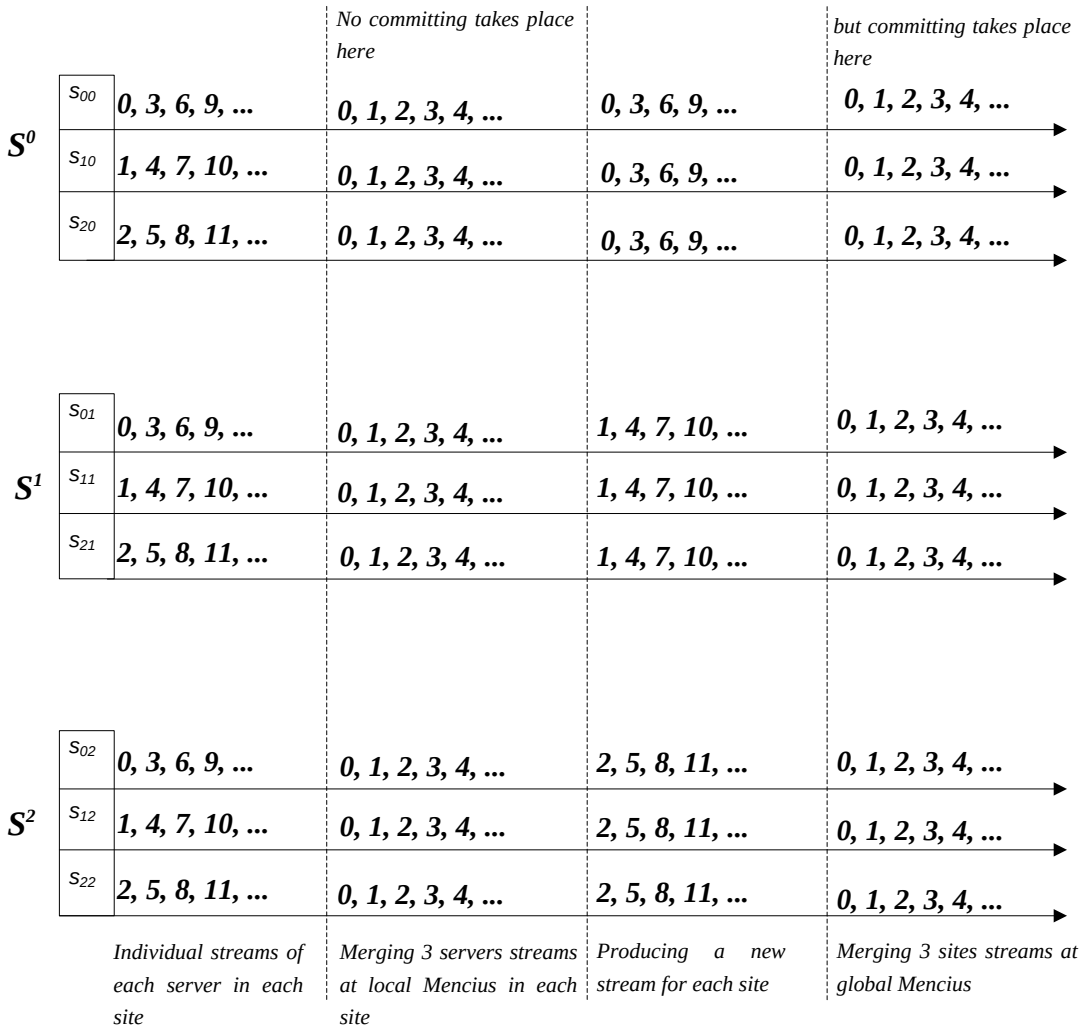

Figure 4.5: Converting streams from local *Mencius* to global *Mencius*

Figure 4.6 explains the steps of forming global commit streams:

Step 1- each site will form its local stream.

Step 2- each site will convert its local stream to produce its part of the global commit stream, using the following equation *iN+SI.*

Step 3- each site will merge its global stream with the ones received from the rest of sites to produce global commit stream.

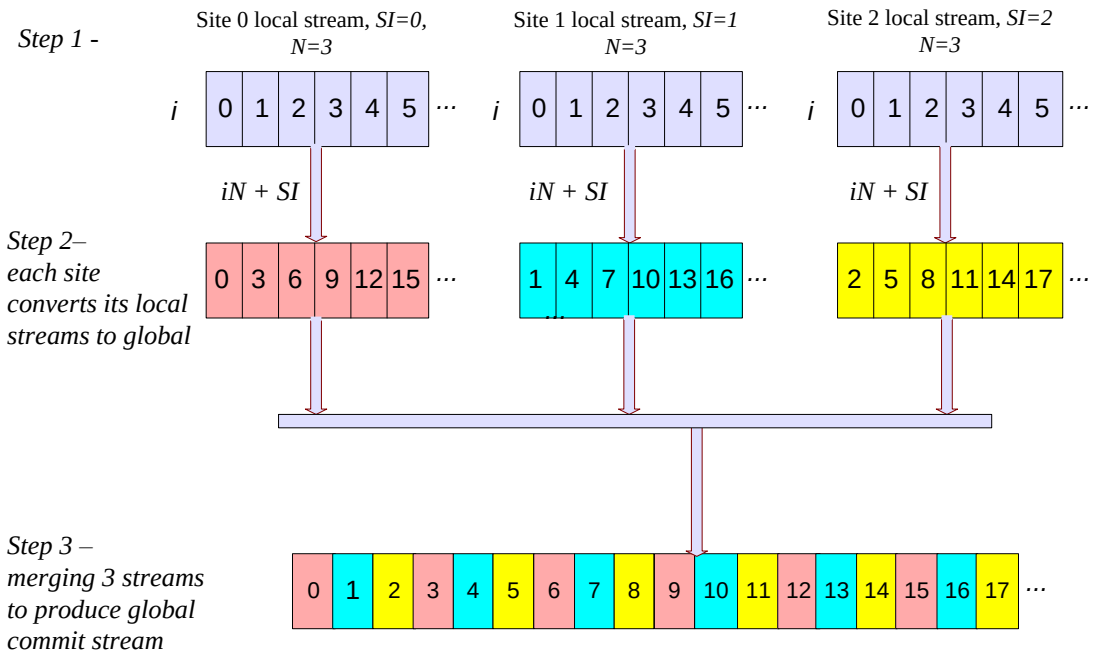

Figure 4.6: forming global commit stream

We can set a general formula that can be used to calculate a global index number for any server at any site as follows:

$$(N * s_{ij}) + S^{SI} + (c * n * N)$$

Where:

N : the number of sites

n : the number of servers

c instance number , $c \in \mathrm{N}_0$

s_{ij} server, where $s_{ij} \in \{0, \ldots, n-1\}$.

S^{SI} site, where $S^{SI} \in \{0, \ldots, N-1\}$.

[Mencius]^N protocol is built on top of the abstraction of *Mencius*. Some of the issues that may encounter *Mencius* can also encounter *[Mencius]*^{N}. It is known that the problem of idle server that faces *Mencius* (which was solved by sending *skip* message either explicitly or implicitly by the idle server), could also face *[Mencius]*^{N}, where a whole site might go idle. This can happen when all servers in a site go idle (no requests are received from all the clients connected to that site), which causes that site to go idle as well. All other sites will not be able to commit any learned messages as long as there are any idle sites. Solving this problem means that the idle site must be induced to generate learn messages with *no_op* (skip) values. We find that there is a necessity to set new Rules to accommodate the changes needed by *[Mencius]*^{N} protocol to address the aforementioned problem.

Rule 1a- Each site S^{SI} maintains its own global index number I_{SI}, site S^{SI} suggests the client's request to consensus global instance I_{SI} and updates I_{SI} to the next global instance. Each request will be mapped to its global instance number.

Rule 2a – Each server s_i maintains its next global instance I_{SI}. Upon receiving a learn message from another site S^{SI}, server s_i compares its own I_{SI} with the other I_{SI} of the received learn message. If the received one is greater than I_{SI} of the local server , then *SKIP* message will be sent locally for that instance and updates its I_{SI} (global) index to the next instance. The solution provided here indicates that the slow sites will be forced to speed up and follow the site with the highest rate of requests.

4.5 Protocol design

Figure 4.7 shows that *[Mencius]*^{N} protocol is based on the abstraction of *Mencius*, and *Mencius* is based on the abstraction of Paxos. This reflects the behaviour of the new protocol.

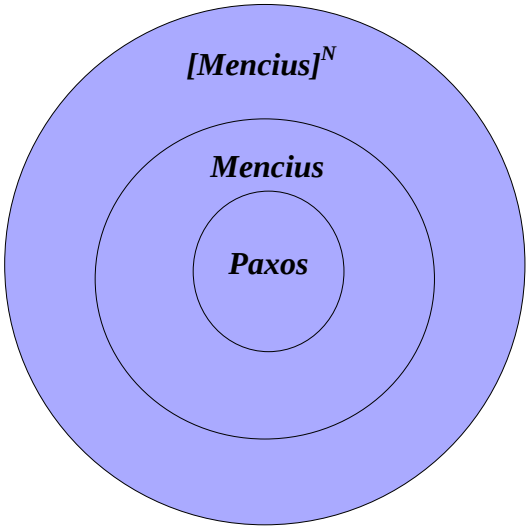

Figure 4.7: *[Mencius]*^{N} and its level of abstraction

[Mencius]^N consists of *N* sites, each site has *n* servers that are publicly ranked. Let this ranking be s_1, s_2,…, s_{2f+1} and be known to all set of servers. All servers within a site are normal servers (they just execute local *Mencius*) except one distinguished server, which is called *Site Speaker*. This *Site Speaker* has two extra jobs: first, to transfer local streams of instances to a global one and second, to communicate with other sites. This is what differentiates site *Mencius* in *[Mencius]*^{N} system from the *Mencius*. If the *Site Speaker* crashes, another server takes the position as a new *Site Speaker*. The following are some facts about *Site Speaker:*

1- At initialisation stage addresses of all servers from each site must be registered with all other sites.
2- At initialization time, the server with the highest rank s_1 will be the *Site Speaker* at all sites

3- Each server registers the name of the local *Site Speaker* and also the ones from other sites as well.
4- Communication between different sites goes through *Site Speaker*s only.
5- *Site Speakers* just send and receive to/from other *Site Speakers*.

All sites are executing *Mencius* in the background, which we call local *Mencius*. Two aspects of *Mencius* are important to our protocol. The first one is the stream of instances produced by each *Mencius* and how that stream is merged with other streams to form what we called global stream or $[Mencius]^N$ stream. The second one is how a crashed or suspected *Site Speaker* is replaced.

The abstraction of *Mencius* is broken down into two different abstractions: the abstraction of *Mencius* stream and the abstraction of replacing a suspected or crashed server which is inherited from Paxos. *Mencius* solves the problem of replacing a suspected or crashed server in terms of revocation only, as all servers are equal and doing the same job. However, this is not the case in $[Mencius]^N$ because servers in each site are divided into two groups, according to their function. One group consists of one server which is called *Site Speaker*. The second group consists of the rest of the servers which is called the normal group. Suspicion of failure or crash in the normal group is treated in terms of revocation, exactly as in *Mencius*, which guaranties that local *Mencius* can make progress. Nevertheless, suspicion of failure or crash of the *Site Speaker* should be solved in two directions: one direction is revocation as the other group, and the second direction is to replace the *Site Speaker*.

Suspicion of failure or crash of the *Site Speaker* affects $[Mencius]^N$, as this distinct server plays a crucial role in the execution of the protocol. $[Mencius]^N$ relies on *Site Speakers* communication to make progress. The replacement of *Site Speaker* will ensure that $[Mencius]^N$ can make progress as well. The following two sections will illustrate first the normal work of the protocol and second how the protocol will behave in case of *Site Speaker* crash or false suspicion.

4.5.1 Normal work of $[Mencius]^N$

The normal execution of the protocol goes through two stages; the first stage is at local level and the second one is at global level.

The first stage of the protocol takes place within a site, which we call local *Mencius*. Local *Mencius* runs normally as described in the previous chapter. Every server generates its own stream, then all streams are collected and merged together to produce local stream or site stream. Local *Mencius* site stream instances will be reported to other sites in the correct order.

In the second stage, or at the level of global *Mencius*, sites communicate with each other through a *Site Speaker*. It is known that the current *Site Speaker* might crash at any arbitrary time while it is reporting its local stream. This may create a problem for the new *Site Speaker* when it takes over as a new *Site Speaker*. The problem created for the new *Site Speaker* is that, from which point it must start reporting local learned messages to other sites. So to facilitate the replacement of the current *Site Speaker* with a new one we need to state how it is going to report local learned messages to other sites. We will suggest two ways to be adopted to address this issue.

First, as the current *Site Speaker* reports learned messages to other sites, it will also divide the local stream to a number of ranges. For example, every range consists of 20 messages and as soon as the current *Site Speaker* completes reporting the last message of that range to other sites, it will send a special message to all local servers informing them that this range was sent successfully to other sites. Every time a range is completed, it will be reported to local servers, range by range, as long as the current *Site Speaker* is alive.

The other way to address this issue is that the current *Site Speaker* will report learned messages to other sites without using any ranges.

This is how the protocol executes when *Site Speakers* are not falsely suspected or crashed. However, false suspicion or crash of the distinguished server can happen in any arbitrary time and will be explained in the next section.

4.5.2 *Site Speaker* change in $[Mencius]^N$

Communication between sites takes place between *Site Speakers* only which might become a single point of failure. The simple solution is to allow all servers from one site to send learn message to all servers from other sites. This redundancy will surely overcome the problem of *Site Speaker*, however, this solution will overwhelm the network with redundant messages which, in turn, will increase message complexity. Every site will receive n^2 copies of the same message.

We propose the following solution to overcome these problems. At the initial stage, addresses of all servers from each site are registered with other sites. *Site Speaker* from each site is known to other sites as the server with the highest rank, and also known to local servers as well. *Site Speakers* will exchange learnt messages, which means communication is *1 to 1*. As soon as *Site Speaker* receives a learnt message, then that message will be multi-casted to all servers within its own site, *1 to n*.

While a *Site Speaker* tries to send a learnt message to another *Site Speaker,* it detects the loss of TCP connection. It will then continuously try to send that message to its destination. If there is a temporary drop in the connection, messages will not reach the destination until a connection is eventually restored, unless the destination *Site Speaker* has crashed. In that scenario, a new *Site Speaker* will be selected, to which all communication is redirected. In the following section we will explain in detail how a new *Site Speaker* is selected and then installed.

4.5.3 Installing a new *Site Speaker*

Site Speaker has a special role to play in $[Mencius]^N$ protocol, which makes it a cornerstone in the building of the protocol. Generally speaking, *Site Speaker* is a normal server that has more jobs to do than other servers. This server is prone to crash, false suspicion or overload of work. With those problems facing the *Site Speaker*, the system must be provided with the right mechanism to replace the current *Site Speaker*. Those problems can be classified according to the reaction taken by the system into two groups: one group consisting of crash and false suspicion, the second group consisting of overload. When the current *Site Speaker* suffers from crash or false suspicion, then the system will enforce a new *Site Speaker* which means the correct servers will force the existing *Site Speaker* to retire. Nevertheless, when the current *Site Speaker* is overloaded, it then asks for a replacement. This is done for the sake of load distribution balance. Both cases will be explained in more detail in the following sections.

4.5.3.1 Enforcing a new *Site Speaker*

All s servers ($s_1, s_2, \ldots, s_{2f+1}$) are publicly ranked, and defined therefore by the sequence in which they are to be the *Site Speaker*. Let this ranking be $s_1, s_2, \ldots, s_{2f+1}$

and be known to all set of servers. At initialization time server s_1 will be the *Site Speaker* at all sites and as soon as s_1 is crashed and successfully revoked, then s_2 will take over as the new *Site Speaker*. If s_2 was also crashed, then s_3 will be the new *Site Speaker* and so on.

All servers in a site register the current *Site Speaker*. When that *Site Speaker* crashes or is suspected of being failed, some server will eventually revoking it. The majority of servers will participate in that revocation and notice that some servers succeeded in that revocation. The next server in ranking will eventually learn that the current *Site Speaker* has been successfully revoked and it will install itself as a new *Site Speaker*. Let us assume s_i is the current *Site Speaker* and was successfully revoked. If s_{i+1} is correct, it will eventually learn the outcome of this revocation and take over as a new *Site Speaker*. Selecting a new *Site Speaker* will adopt the same mechanism of Chandra and Toueg protocol, where the choice of a new leader (*Site Speaker)* is based on the rotating coordinator paradigm. Failure detector oracle $\Diamond S$ provides its process with a list of suspected processes that have crashed; the choice of the new leader *(Site Speaker)* is done in a round-robin fashion.

The current *Site Speaker* might be suspected or crash at any time while it is doing the job of reporting learnt messages to other sites. However, the server that will take over as a new *Site Speaker* will not be able to decide from which learnt message it must report to other sites. There are two ways to solve this problem that was proposed at section 4.5.1.

If we adopt the first solution in which the current *Site Speaker* divide the local stream to a number of ranges, then when another server takes over as a new *Site Speaker*, it will pack all learn messages (starting from the first message that comes after the last range up to the last learn message before crash or suspicion) into one message and send it to all other sites. This might cause a message to be sent more than once (by the old and the new *Site Speaker*), however, this will be discovered by other sites and considered as a duplication, hence discarded. However; if we adopt the second solution then, as soon as the old *Site Speaker* is replaced by another server, the new *Site Speaker* starts reporting learnt messages (starting from the last learnt message before crash or suspicion) to all other sites. When the receiving site finds out that there are some missing messages, it will then ask the new *Site Speaker* about the learn messages not yet received.

Correct servers eventually learn about the replacement of the existing *Site Speaker* and also know that the next server in ranking (if it is correct) will be the new *Site Speaker*. Any change in the *Site Speaker* must be registered by the correct servers. This update of the state of the system is necessary because false suspicion and crash are inevitable actions that can happen to any server including the *Site Speaker* at any arbitrary time. All servers in a site must have full information about the state of the system in order to be ready to act as a *Site Speaker* if necessary.

False suspicion is inherited from unreliable failure detectors, so if the old *Site Speaker* was falsely suspected or recovers from a crash and learns that some other server is acting as a *Site Speaker,* it will then retire from acting as a *Site Speaker* and join the group to continue the normal job as an ordinary server.

4.5.3.2 Asking for a *Site Speaker* replacement

In addition to executing local *Mencius* (coordinating its client requests), *Site Speaker* has to send local site stream to all other sites, receive streams from other sites and multicast them to local servers. This reflects the fact that network traffic from or to the *Site Speaker* is much higher than its counterparts in the site. The problem is compounded if it has active clients. In such circumstances, much of the bandwidth will be consumed by the *Site Speaker* which produces unbalanced communication pattern.

The second issue is that as *Site Speaker* receives higher number of messages, there is a higher demand for processing power than other servers. Even though messages are not really processed, receiving and sending messages continues to consume processing power.

In trying to get a more balanced communication pattern and better CPU system utilization, the current *Site Speaker* will seek the server with the lowest load and hand to it the *Site Speaker* responsibility.

Every correct server has to participate in executing local *Mencius* to coordinate its own requests. The load of ordering requests for each server depends on the number of requests received in a certain period of time in relation to other servers in the same site. Servers with very active clients will propose real messages, others with idle clients will generate skip only, and a third group, with a moderate load, will produce both. Individual streams from each server will be received by all correct servers in

that site. This will enable the *Site Speaker* to calculate the load distribution of each server. That can be achieved by counting the number of real messages *(NRM)* and skip messages *(NSM)* received at a certain period of time (from t_1 to t_2) from all correct servers, then subtracting both values (*NRM – NSM).* The results from all servers tells the *Site Speaker* which server has the highest load and which one has the lowest load.

The *Site Speaker* periodically will make these calculations and analyze the collected data. The servers with positive results means that more real proposals than skip messages are produced. Zero means the same number of real proposals and skip messages are generated, and servers with negative results mean that they have inactive clients.

The *Site Speaker* will choose the server with the smallest negative number and asks it to be the new *Site Speaker*. The procedure of this replacement goes through the following steps:

1. First; the current *Site Speaker* finds out the server with smallest negative number
2. It sends a special proposal with *no_op* value and the ID of the target server
3. If targeted server is correct eventually will receive that message and respond with accept message.
4. The original *Site Speaker* waits to receive accept message from the majority.
5. The *Site Speaker* after receiving from the majority generate learn message to inform the whole group about the handing over of the responsibility of *Site Speaker* to a new server.

When the local load is evenly distributed or the current *Site Speaker* has low load, then there is no need for any change and the current *Site Speaker* will continue functioning as a *Site Speaker*.

4.5.3.3 How a new *Site Speaker* starts its job?

As soon as some server successfully installs itself as the new *Site Speaker,* it starts communication with other sites by multicasting learn messages to all servers, *1 to n,* which will inform all servers at other sites about *Site Speaker* change. The

process of multicasting *1 to n* will continue until the new *Site Speaker* receives learn message from other *Site Speakers.* It will then switch from *1 to n* communication to *1 to 1*. The main reason for *1 to n* communication, started by the new *Site Speaker*, is to eliminate the deadlock that might be created by the crash of more than one *Site Speaker* at the same time. To explain this problem, let us assume that s_{11} is the current *Site Speaker* of site S^1 and s_{12} is also the current *Site Speaker* of site S^2, both of them crashed at the same time. Let us assume that s_2 was selected as a new *Site Speaker* of both sites. Each s_2 knows in advance that s_1 is still the *Site Speaker* and has no knowledge about the crash at the other site. If both s_2 use *1 to 1* communication, then each one will try to send learn messages to s_1 at the other site, which will create a deadlock as both s_1 already crashed and was replaced by s_2. To avoid such situation we resorted to *1 to n* communication as a starting point, which will introduce the new *Site Speaker* to all servers at all sites.

4.6 *Site Speaker* Algorithm

When the current *Site Speaker* crashes some correct server will discover that by its FD. Failure detector oracle provides its process with a list of suspected processes that have crashed; the choice of the new leader *(Site Speaker)* is done in a round-robin fashion. All servers are publicly ranked and defined therefore by the sequence in which they are to be the *Site Speaker*. The new *Site Speaker* should takeover according to the following steps:

1- The next server in ranking will eventually learn that the current *Site Speaker* has been crashed and installs itself as a new *Site Speaker*.
2- It starts reporting learnt messages (starting from the last learnt message before crash or suspicion) to all other sites.
3- The new *Site Speaker* starts communication with other sites by multicasting learn messages to all servers, *1 to n.* The process of multicasting *1 to n* continues until the new *Site Speaker* receives learn message from other *Site Speakers,* then it switches from *1 to n* communication to *1 to 1*.
4- If the receiving site finds out that there are some missing messages, then it asks the new *Site Speaker* about the learn messages that are not received yet.

4.7 Summary

This chapter presented the design and the implementation details of *[Mencius]*^{N} protocol. The concept of structuring and designing the protocol as a multi-cooperative Mencius produced a crash-tolerant protocol. Paxos is used as the core protocol to solve *Total Order* problem as one form of agreement problem. The execution of consensus protocol was kept as an internal issue within each site.

This novel idea of building the new protocol on two levels created an environment from which the traffic of exchanged messages on wide-area network was reduced. The correctness of the new protocol derived from the correctness of Paxos and Mencius as well, with the safety and liveness requirements both being preserved. The benefit of reducing the traffic of exchanged messages for each instance on wide-area network and restricting it to only one message has many advantages. Firstly, this led to reduction of bandwidth consumption. Secondly, latency of committing requests was decreased as well.

There are other benefits achieved from this novel idea of building the system of multiple standalone Mencius'. Firstly, no more clients are blocked and secondly, the threshold of saturation is increased.

Chapter 5

Experiments and Results

5.1 Introduction

The objective of this work is to design and present a protocol that has a better performance than *Mencius* over WAN. It requires the recognition of the challenges faced by *Mencius* and the costs required to achieve the improvements we proposed. This chapter is dedicated to comparing the performance of our protocol $[Mencius]^N$ with *Mencius* and to present the reader with our findings that are extracted from the data collected from the experiments.

The stage of evaluating and analysing the performance of both protocols has gone through an extensive testing period. This included the establishment of a series of comprehensive experiments designed to evaluate quantitative system performance in a wide range of parameters. In this chapter the reader is first presented with the environment of the experiment and its settings and then second, shown the evaluation of this work through the results. Finally, the chapter concludes with a summary.

5.2 Experimental Environment

The purpose of these experiments is to compare the performance of our protocol $[Mencius]^N$ with *Mencius*. Both protocols are implemented in Java and are evaluated on a single network cluster that provides enough machines to test the two protocols. Each machine is a 1.86 GHz Intel Core (TM) 2 PC with 2.0 GB memory running Fedora 12. In *Mencius* we use 3 machines, each of which represents one site. In $[Mencius]^N$, $N=3$, 9 machines are used, each group of 3 machines representing one site. TCP is used as the transport protocol.

In local-area networks, machines are automatically and periodically synchronized. This aims to guarantee that they have minimal time drift. Time drift is measured before and after carrying out any experiment, in order to guarantee that all the machines have almost the same time clock.

For measurement purposes (Note: this is not a design requirement), Each experiment run consists of three parameters: **link time delay, request arrival time interval, and number of requests** generated by each client during each run.

5.2.1 Link time delay and bandwidth

In order to emulate wide-area network, a virtual link is created between each two sites using DummyNet [Riz97, CR09]. DummyNet is a tool used to enforce different time delays and bandwidth [MJM08, MF09, JS08, CR+09]. The link bandwidth values used are 10 Mbps and 20 Mbps. As the two link bandwidths produce similar results, we present only 10 Mbps in this work.
With regards to link time delay, three different classes are used:

Class I – In this class we carry out the experiment without using DummyNet. In this case, time delays of the virtual link are dictated by the local-area network, which are measured and found to be approximately 3ms.

Class II – In this class we use DummyNet to time delays for all three links. We experiment with one-way fixed-delay settings of 25 ms, 50 ms, and 100 ms for each experiment.

Class III – In this class we use Mixed-delay settings which are taken from a real experiment [CR+09]. This experiment investigates how Internet delays vary in the context of assessing timeliness of Web Services from a user's perspective. To provide a comprehensive assessment, the experiment uses five clients deployed in different places over the Internet: Frankfurt (Germany), Moscow (Russia), Los Angeles (USA) and two clients in Simferopol (Ukraine), all using different Internet service providers. The experiment reports the response time between Newcastle and these five clients. Our intention is to use these traces as a time delay for each message sent out between sites. Because of the limitations of DummyNet, which only gives the capability of enforcing one time delay for each or all links but not time delay for individual messages, a different approach is required. It is decided to use three different time delays, each link with its own delay.

We take average response time of three cities (Frankfurt, Moscow, and Los Angeles) with Newcastle. From that we calculate one-way time delay of each city.
Time delay of 110 ms between Newcastle and Frankfurt is chosen as one-way time delay for the first link. Time delay of 533 ms between Newcastle and Moscow is used

as one-way time delay for the second link. For the third link, a time delay of 577 ms between Newcastle and Los Angeles is used as one-way time delay.

5.2.2 Request arrival time interval

In the experiments, different arrival time intervals (AI) between requests are used. Arrival time interval values are chosen from uniform distribution (20ms, 38ms, 75ms, 150ms, 1000ms and 10000ms). Other experimental results, using AI values < 20ms, are reported in one section, but not reported in others, as latency has significantly increased for both protocols.

5.2.3 Number of requests

We carried out our experiments using 30000 requests per client for the following arrival time interval values 20ms, 38ms, 75ms, 150ms, 1000ms. For the 10000ms arrival time interval, 10000 requests per client is used.

5.3 Experimental settings

In order to compare *Mencius* with $[Mencius]^N$, we design each protocol with two layers. The higher layer represents the service provided while the lower layer represents the ordering protocol. We use a simple service that assigns an order to each request received from the clients. Our design and implementation mainly focuses on ordering protocol. Committing order is represented by storing requests into a file. The log file is used to verify that all servers learn and commit the same client request in the same sequence which shows that sites are in consistent state.

5.3.1 Number of experiments

The number experiments (for each run) that we carry out for all time intervals are ≥ 5, except for 10000ms, in which we carry 3 experiments only. The results reported in our thesis represent the average of all these experiments. All the experiments are run over week ends because The University's internal network activity is minimal at this time.

5.3.2 Message length

Original message length is 72 bytes; these bytes represent different parts of the message, such as client ID, message sequence, server ID, etc. An extra payload of 1388 bytes is added to get the size of TCP data to 1460 bytes. The main reason for not using more than 1460 bytes is to prevent segmentation at the Transport Layer. The wisdom behind the use of a long message is to prevent the system adding any time delay by Nagle's algorithm [NAG84]. Long message is used in section 5.4.1 for throughput analysis and in section 5.4.2 for latency analysis.

Message length is used for optimizing the protocol. We exploit the short size of the request which is 72 bytes by batching multiple requests into a single message. The usage of batching short messages and their evaluation is found in section 5.6.

5.3.3 Clients engineering

In *Mencius,* each site consists of one server and each server is associated with its own client. However, in $[Mencius]^N$, each site consists of 3 servers and each server is associated with its own client. This results in 3 clients for one site. Instead of using separate physical machines for each client, the client is built inside each server instead. This solution is chosen to reduce the number of machines needed to test both protocols, especially with $[Mencius]^N$.

In order to get the same rate of requests from each site for both protocols, the following assumptions are made:

1. In *Mencius,* each client connects to one site generating 30000 requests.
2. In $[Mencius]^N$, three clients connect to one site generating 30000 requests, which means each client generates 10000 requests only.

The above assumptions imply that both protocols are tested under the same load and the same circumstances.

We start by analysing the throughputs of both protocols using different time intervals between requests and different time delays of our virtual link that represents a wide-area network. The latency of committing requests under different settings as mentioned earlier is then evaluated. Next, the effect of optimization that can be introduced to the protocol to enhance performance is evaluated. Finally, revocation and comparison between *Mencius* and revised version of *Mencius* is evaluated, which is presented in chapter 3.

5.4 Evaluation for large requests

5.4.1 Throughput

To measure throughput, Long Message is used and five sets of experiments, each one having a number of runs, are conducted. These experiments differ on the time delay of the virtual link. Each run has its own time interval between requests generated by clients. The following time interval values are chosen from uniform distribution (20ms, 38ms, 75ms, 150ms, 1000ms and 10000ms). The result of all experiments were tabulated and graphed. Experiment results for throughput are presented in Tables 5.1 to 5.10 and Figure 5.1 to Figure 5.5. The results indicate clearly that the time delay of the virtual link has no effect on throughput and all experiments produce almost the same throughput.

Results indicate that protocol *[Mencius]*^{N} has higher threshold of saturation over *Mencius.* This is apparent when the time intervals between consecutive requests are decreased. To elaborate on this point, at a rate of 1000ms/site, the throughput of both protocols was 3 requests per second. As an example, results are presented in Table 5.1 and Table 5.2, and Figure 5.1, those two tables and one figure are representing LAN time delays.

[Mencius]^N	
Arrival Intervals ms	Throughput
20	140
38	76
75	39
150	20
1000	3
10000	0.3

Table 5.1: throughput *[Mencius]*^{N}

Mencius	
Arrival Intervals ms	Throughput
20	124
38	71
75	38
150	19
1000	3
10000	0.3

Table 5.2: throughput *Mencius*

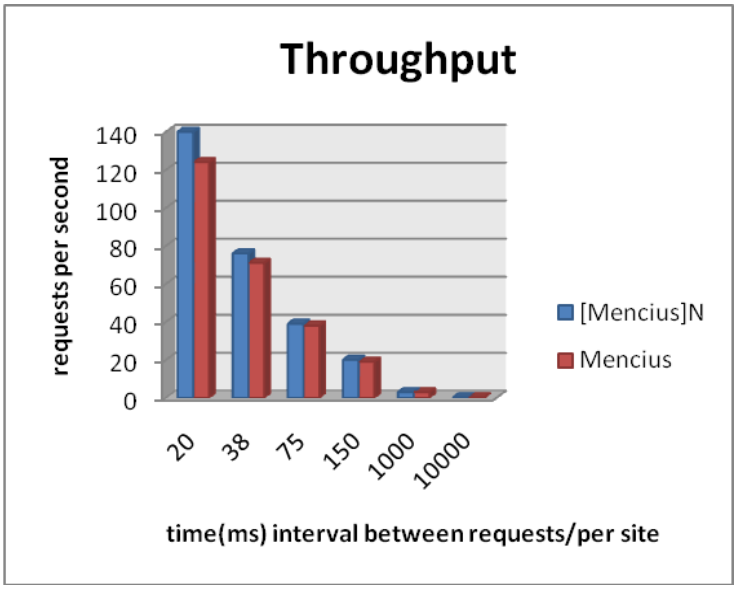

Figure 5.1: Experiment 1; time delay of LAN, no DummyNet.

At a rate of 150ms/site, the throughput of both protocols is approximately 20 requests per second. However, at rate of 38ms/site, some important differences are observed. Here, the throughput of protocol *[Mencius]*^{N} is 76 requests and for *Mencius*, 71. By decreasing the time interval and reducing it to 20ms/site, the throughput of protocol *[Mencius]*^{N} is 139 requests and for *Mencius,* 124. This indicates a clear difference between the two protocols.

At lower rates of request, both protocols produce the same throughput. However, with a higher rate of request, significant differences in the throughput of both protocols are observed.

This can be attributed to a higher number of nodes at each site for our protocol *[Mencius]*^{N}. Each site consists of *3* nodes, which increases their ability to cope with a higher rate of request, but having one node at each site for *Mencius* limits it to coping with a higher rate of requests. We can conclude that increasing the number of machines in the new protocol translates to a higher processing power in comparison to one machine in *Mencius*. Results for other link time delays are shown in Table 5.3 to Table 5.10 and Figure 5.2 to Figure 5.5.

[Mencius]^N -25	
Arrival Intervals ms	Throughput
20	139
38	76
75	39
150	20
1000	3
10000	0.3

Table 5.3: throughput *[Mencius]*^{N}-25

Mencius -25	
Arrival Intervals ms	Throughput
20	124
38	71
75	38
150	19
1000	3
10000	0.3

Table 5.4: throughput *Mencius*-25

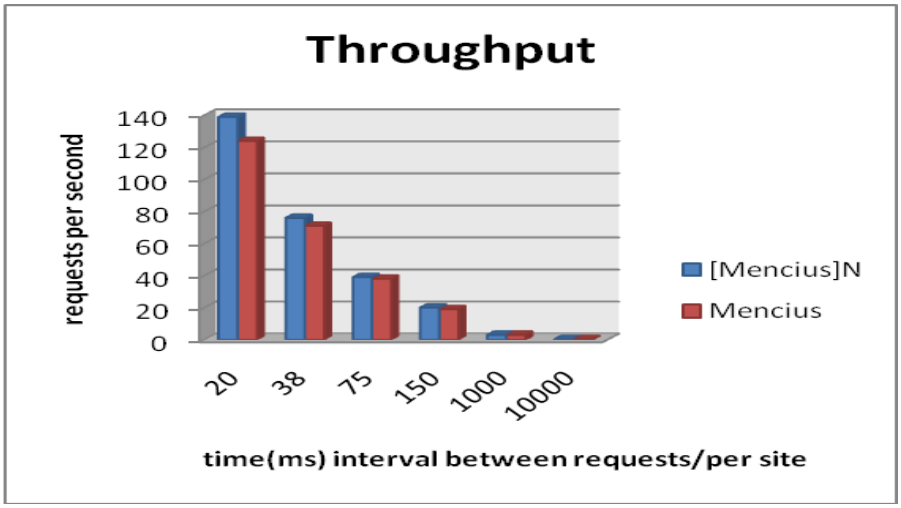

Figure 5.2: Experiment 2; one-way delay of 25ms.

[Mencius]^N -50	
Arrival Intervals ms	Throughput
20	139
38	76
75	39
150	20
1000	3
10000	0.3

Table 5.5: throughput *[Mencius]*^{N}-50

Mencius -50	
Arrival Intervals ms	Throughput
20	124
38	71
75	38
150	19
1000	3
10000	0.3

Table 5.6: throughput *Mencius* -50

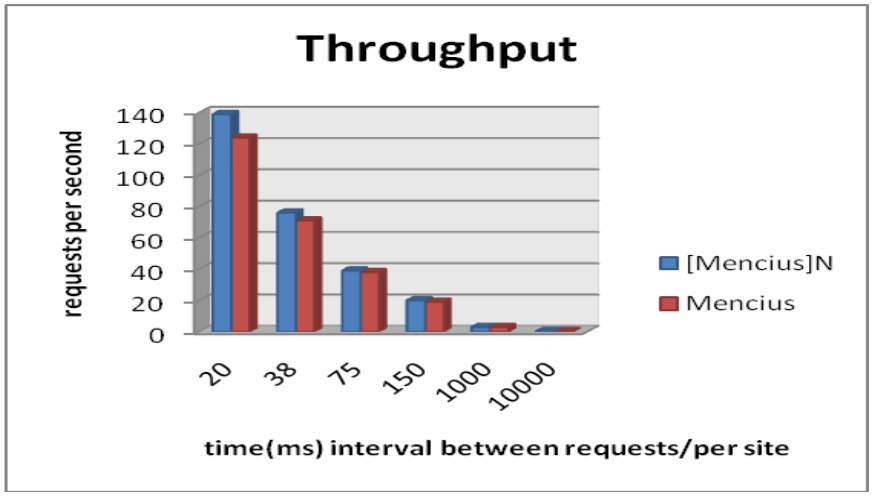

Figure 5.3: Experiment 3; one-way delay of 50ms.

[Mencius]^N -100	
Arrival Intervals ms	Throughput
20	138
38	75
75	39
150	20
1000	3
10000	0.3

Table 5.7: throughput *[Mencius]*^{N}-100

Mencius -100	
Arrival Intervals ms	Throughput
20	124
38	71
75	38
150	19
1000	3
10000	0.3

Table 5.8: throughput *Mencius*-100

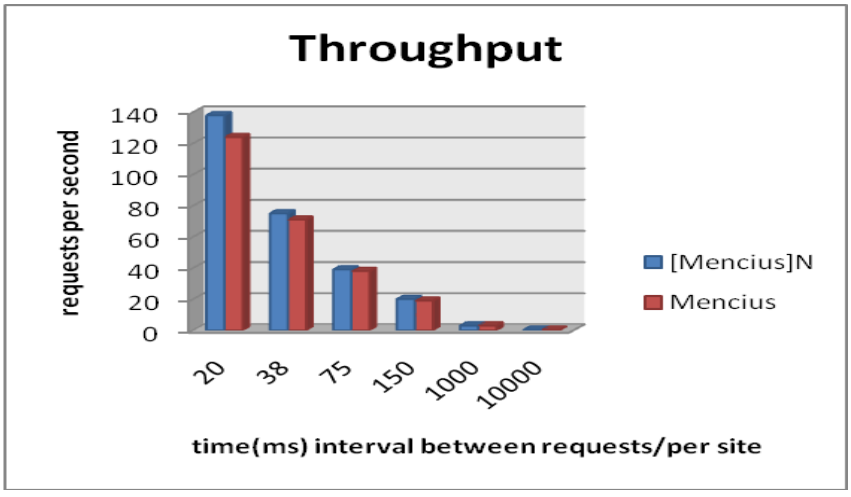

Figure 5.4: Experiment 4; one-way delay of 100ms.

[Mencius]^{N} -D	
Arrival Intervals ms	Throughput
20	138
38	76
75	39
150	20
1000	3
10000	0.3

Table 5.9: throughput *[Mencius]*^{N}-D

Mencius -D	
Arrival Intervals ms	Throughput
20	120
38	70
75	37
150	19
1000	3
10000	0.3

Table 5.10: throughput *Mencius*–D

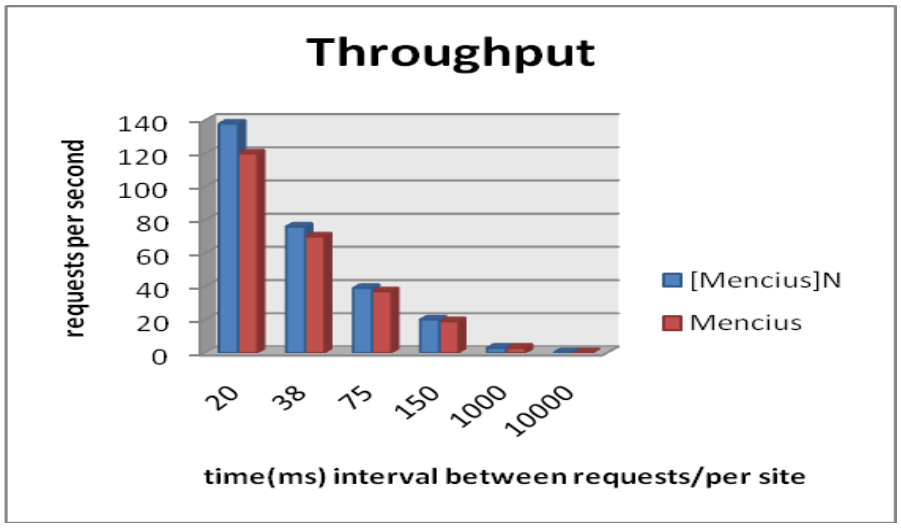

Figure 5.5: Experiment 5; Different one-way time delays.

5.4.2 Latency

The same results of the experiments that are used to measure throughput are also used to measure latency. The same settings, the same number of experiments and the same number of runs are used. Each experiment is distinguished with its own time delay of the virtual link that represents wide-area network. In the first experiment, the time delay of the virtual link is inherited from local-area network delays, so there is

no use of DummyNet. The other four experiments are distinguished with different time delays. The results of all experiments are presented in the form of tables and charts, with each experimental result presented in two tables and one chart.

Unlike throughput, the time delay of the virtual link has a significant effect on latency. The experiment is divided into two groups; one set of experiments belonging to the first group and the other four sets of experiments belonging to the second group. The set of experiments characterized with link time delay inherited from local-area network are placed into the first group and the other four experiments with delay enforced by DummyNet are placed into the second group.

Two latency values, max-Latency and min-Latency, are distinguished. Max-Latency is the time elapsed starting from the time a request received by a server to the time that request is committed by *the last server* in the system. Min-Latency is the time elapsed starting from the time a request is received by a server to the time that request is committed by *the first server* in the system.

We start by explaining latency in relation to the first group. The results of our first set of experiments belonging to the first group are presented in Table 5.11, Table 5.12 and Figure 5.6. The time delay of the virtual link in this experiment is inherited from local-area network delay. *Mencius* has lower latency (both max-Latency and min-Latency) compared to $[Mencius]^N$.

Protocol $[Mencius]^N$ suffers higher latency because its execution goes through two levels. Within each site, *Mencius* at a local level was executed within each site. At the global level, the *Site Speaker* exchanges global stream with other *Site Speaker*s. This reflects that the *Site Speaker* consumes more processing time and an extra message is needed to report the outcome of local level of the protocol to the global level of the protocol for each instance. As an example, for the time interval of 20ms/site, the max-Latency of protocol $[Mencius]^N$ is approximately 70 ms compared to 53 ms for *Mencius*. For min-Latency of protocol $[Mencius]^N$, the latency is approximately 62 ms compared to 48 ms for *Mencius*. From a latency perspective in such circumstances, *Mencius* has better performance than $[Mencius]^N$.

[Mencius]^N		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	69.89	62.07
38	68.67	62.54
75	69.06	61.39
150	66.22	61.59
1000	66.71	54.33
10000	68.19	55.58

Table 5.11: latency *[Mencius]*^{N}

Mencius		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	53.03	47.7
38	56.04	51.76
75	56.48	52.94
150	52.36	44.17
1000	53.62	33.98
10000	58.31	32.96

Table 5.12: latency *Mencius*

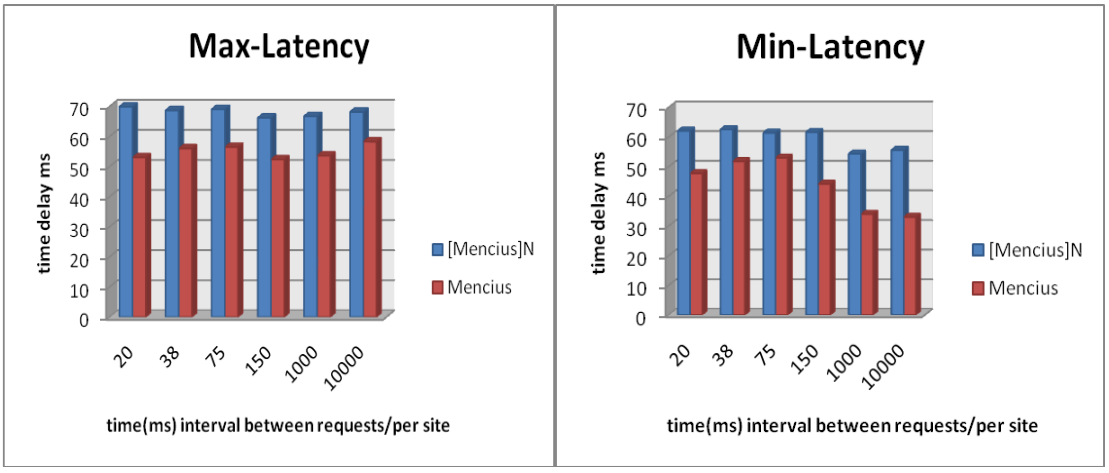

Figure 5.6: Experiment 1; time delay of LAN, no DummyNet.

Now we are trying to investigate and analyse Latency in relation to the second group of our experiments. Four sets of experiments are included in the second group with time delay of the virtual link in this group enforced by DummyNet tool. The results of these experiments are shown in Table 5.13 to Table 5.20 and Figure 5.7 to Figure 5.10.

We start with the first set of experiments in the second group. This set has 25ms link time delay. Table 5.13 shows the results of six experiments for *[Mencius]*^{N} protocol and Table 5.14 shows the results of another six experiments for *Mencius* protocol. As the delay of the virtual link is increased to 25ms in this set of experiments, results show that *Mencius* suffers from higher latency than *[Mencius]*^{N}. This is because in *Mencius*, all messages needed to execute instances of Paxos must

travel between sites over wide-area network, with each message will incurring an extra delay time. Nevertheless, in *[Mencius]*^{N}, instances of Paxos are executed over local-area network and the delay time of local-area network is not changed. The *Site Speaker* needs to inform other sites about the outcome of each local instance so only this message suffers the delay of 25ms. For each instance in *Mencius*, at least 3 messages suffer extra delay, while in *[Mencius]*^{N}, only one message suffers from that delay. This explains why *[Mencius]*^{N} has lower latency compared to *Mencius*. Figure 5.7 reflects the difference between both protocols.

[Mencius]^N -25		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	98.4	71.7
38	96.8	67.2
75	96.2	65.8
150	94.9	65.1
1000	84.3	60.8
10000	82.9	49.2

Table 5.13: latency *[Mencius]*^{N} 25

Mencius -25		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	120.86	96.79
38	120	98.69
75	131.79	98.46
150	125.4	97.79
1000	97.53	70.23
10000	92.95	61.85

Table 5.14: latency *Mencius* 25

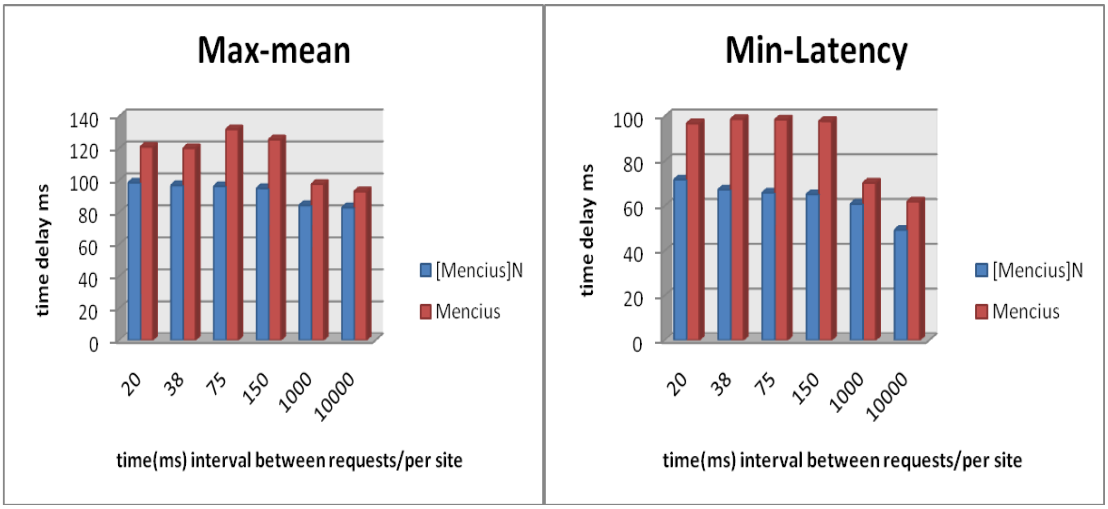

Figure 5.7: Experiment 2; one-way delay of 25ms.

In the second and third set of experiments, the delay time of the virtual link is increased to 50ms and 100ms. The results of these two experiments are found on Tables 5.15 to Table 5.18. From these two sets of tables, one can observe that the difference between the latency of both protocols is increased. It is understood that the higher the delay time of the virtual link, the better performance received from *[Mencius]*^{N} regarding latency. Figure 5.8 and Figure 5.9 reflect these differences clearly.

[Mencius]^N -50		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	119.86	97.49
38	116	91.43
75	117.7	91.74
150	108.05	85.14
1000	106.96	85.24
10000	107.47	81.7

Table 5.15: latency *[Mencius]*^{N} 50

Mencius -50		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	192.36	168.36
38	197.3	160.6
75	207.23	165.39
150	202.63	165.17
1000	172.86	124.17
10000	182.2	130.26

Table 5.16: latency *Mencius* 50

Figure 5.8: Experiment 3; one-way delay of 50ms

[Mencius]^N -100		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	187.7	156.4
38	177.8	140.7
75	173.9	132.7
150	174	130.6
1000	171	127.3
10000	168.6	125.7

Table 5.17: latency *[Mencius]*^{N} 100

Mencius -100		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	397.71	301.77
38	394.58	347.64
75	393.13	311.7
150	379.39	294.96
1000	327.05	245.53
10000	318.99	256.89

Table 5.18: latency *Mencius* 100

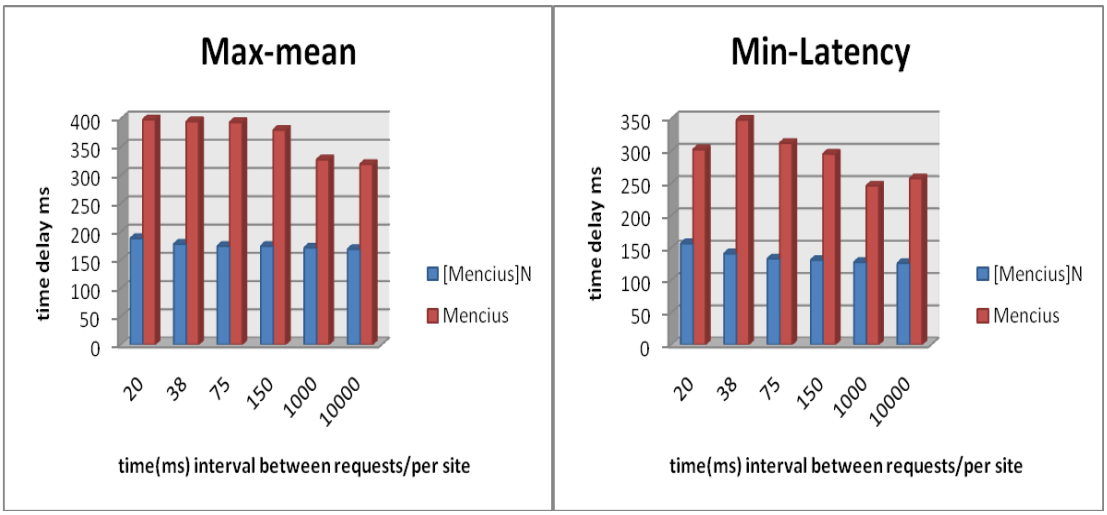

Figure 5.9: Experiment 4; one-way delay of 100ms.

In the last set of experiments carried out to measure latency, three different time delays (which are found in Class III at section 5.2.1) of the virtual link that represents wide-area network, are used. The results of these experiments can be found in Table 5.19 and Table 5.20.

As the values of delay times in class III are significantly higher than class II, which in turn produces a distinct difference between the latency of both protocols. From Figure 5.10, it can be seen that at a low request rate, the difference in latency between the two protocols is around 600ms. However, at higher rates of requests latency difference approaches seconds. These results emphasize that the higher the delay time of the virtual link, the better performance is received from *[Mencius]*^{N} regarding latency.

[Mencius]^N -D		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	1614	1012
38	1412	874
75	1210	781
150	1036	641
1000	820	376
10000	925	374

Table 5.19: latency *[Mencius]*^{N} D

Mencius -D		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
20	2697	2191
38	2429	2050
75	2060	1660
150	1870	1550
1000	1569	1065
10000	1400	1038

Table 5.20: latency *Mencius* D

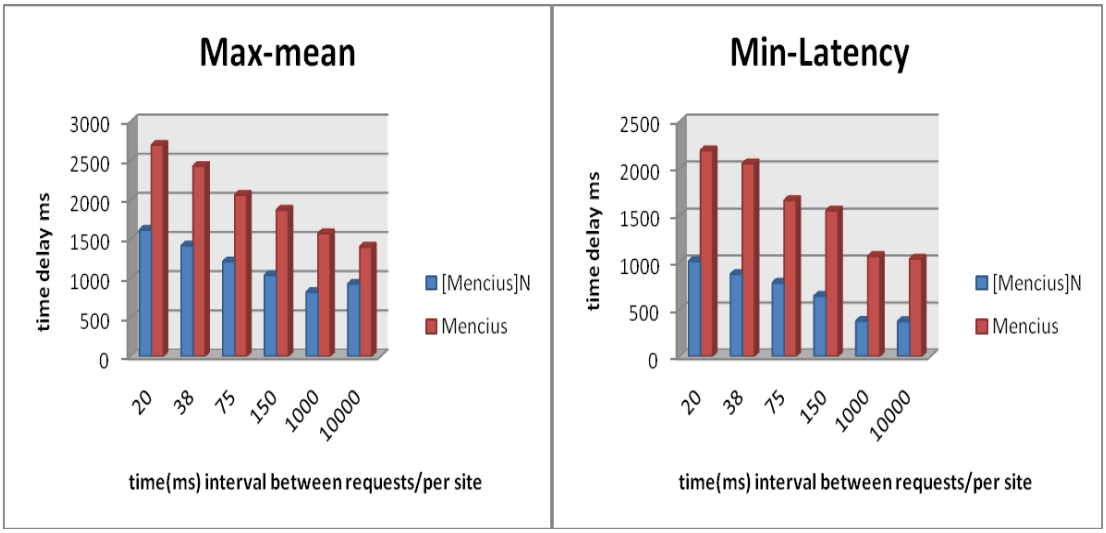

Figure 5.10: Experiment 5; random one-way time delays.

Presenting the results in the form of tables and figures demonstrates the improvement of *[Mencius]*^{N} protocol. However, by presenting performance in terms of *GAIN*, results can be compared more accurately and with improved precision. *GAIN* can be calculated by subtracting any *Mencius* value from the corresponding *[Mencius]*^{N} value. This result is then divided by *Mencius* value and multiplied by 100. The result is the percentage *GAIN*.

GAIN definition:

$$GAIN = \frac{[Mencius]^N \text{ value} - Mencius \text{ value}}{Mencius \text{ value}} \times 100$$

Example, values taken from Table 5.19 and Table 5.20:

$$GAIN_{\text{Max-latency}} = \frac{1614 - 2697}{2697} \times 100 = -40\%$$

From this point on results are presented using the *GAIN* term. The remaining tables and figures can be found in appendix B at the end of this thesis.

5.4.3 *GAIN* summary

The following three tables summarize the conclusion regarding throughput and latency. Table 5.21 shows throughput gain. At arrival interval (AI) of 1000ms and 10000ms, the *GAIN* is zero. At AI of 150ms, the *GAIN* is 5% and reaches 12.5% at AI of 20ms.

Table 5.22 shows max-latency *GAIN* while Table 5.23 shows min-latency gain. In both of these tables, the *GAIN* of the experiments under *Class I* is in favour of *Mencius* but as soon as an increase in virtual link time delay is introduced (*Class II* and *Class III)*, we noticed that the *GAIN* in both forms of latency is reversed and becomes in favour of $[Mencius]^N$.

AI/ms	Class I %	Class II %			Class III %
		25ms	50ms	100ms	
20	13	12	12	11	15
38	7	7	7	6	9
75	3	3	3	3	5
150	5	5	5	5	5
1000	0	0	0	0	0
10000	0	0	0	0	0

Table 5.21 Throughput *GAIN*

AI/ms	Class I %	Class II %			Class III %
		25ms	50ms	100ms	
20	32	-19	-38	-53	-40
38	23	-19	-41	-55	-42
75	22	-27	-43	-56	-41
150	26	-24	-47	-54	-45
1000	24	-14	-38	-48	-48
10000	17	-11	-41	-47	-34

Table 5.22 Max-Latency *GAIN*

AI/ms	Class I %	Class II %			Class III %
		25ms	50ms	100ms	
20	30	-26	-42	-48	-54
38	21	-32	-43	-60	-57
75	16	-33	-45	-57	-53
150	39	-33	-48	-56	-59
1000	60	-13	-31	-48	-65
10000	69	-20	-37	-51	-64

Table 5.23 Min-Latency *GAIN*

5.5 Batching short messages

In order to improve performance, an optimisation to the protocol is suggested in order to find an approach that increases throughput. The idea is to exploit the short size of the request of 72 bytes by batching multiple requests into a single message. Having adopted this approach improves throughput of both protocols but at the expense of latency, which unfortunately increased. The technique applied is to batch a number of requests arriving within a certain period of time to get around size limitations of TCP data which is 1460 bytes.

5.5.1. Throughput

Five sets of experiments are conducted to evaluate throughput. These sets of experiments differ in the time delay of the virtual link. Each experiment has its own time interval between requests generated by clients. The following are the time interval values chosen from uniform distribution (5ms, 20ms, 38ms, 75ms, 150ms). Different arrival time intervals are chosen to the ones used in section 5.4 because there is no sense in applying batches of messages with requests arriving within a second or seconds, such as 1000ms or 10000ms. The results of all experiments are presented in a form of tables and charts which are found in appendix B.

By introducing this technique, the threshold of saturation and throughput improved as well. As explained in section 5.4, the time delay of the virtual link has no effect on throughput. The following four experiments, 20ms, 38ms, 75ms and 150ms produce almost the same throughput as that generated by experiments conducted in section 5.4. By reducing the arrival time interval to 5ms, a difference between the two protocols regarding throughput is observed. *Mencius* produces an average of 329 requests per second while $[Mencius]^N$ produces an average of 439 requests per second. The threshold of saturation increases compared to the usage of a single long message and the throughput of $[Mencius]^N$ improved by around of 33% over *Mencius*. More details about results regarding throughput can be found in Table A.1 to Table A.10 and in Figure A.1 to Figure A.5 in appendix B at the end of this thesis.

5.5.2 Throughput Summary

Table 5.24 summarizes the *GAIN* regarding throughput. At AI of 150ms the *GAIN* is 5%. At AI of 5ms the *GAIN* reaches an average of around 33%.

AI/ms	Class I %	Class II %			Class III %
		25ms	50ms	100ms	
5	40	28	41	30	28
20	13	12	13	11	11
38	7	6	7	6	4
75	3	3	3	3	3
150	5	5	5	5	5

Table 5.24 Throughput *GAIN*

5.5.3 Latency

The implementation of the new technique (batching multiple messages together) enables a better performance regarding threshold of saturation and throughput. Nevertheless, latency increases for both protocols. This can be attributed to the time required to receive a number of messages and the time required to batch them together into a single message. Not surprisingly, despite the increase in latency, it is found that the higher time delays of the virtual link, protocol $[Mencius]^N$ still has lower latency compared to *Mencius.* More details about latency results can be found in Table A.11 to Table A.20 and in Figure A.6 to Figure A.10 in appendix A at the end of this thesis.

5.5.4 Latency summary

The following two tables summarize the conclusions regarding latency. Table 5.25 shows max-latency gain, while Table 5.26 shows min-latency gain. Even though, the difference is not as clear as that in section 5.4.2, lower latency for $[Mencius]^N$ is achieved.

AI/ms	Class I %	Class II %			Class III %
		25ms	50ms	100ms	
5	-1	-6	-12	-23	-19
20	1	-7	-11	-19	-14
38	1	-12	-7	-27	-22
75	-1	-14	-8	-25	-24
150	4	-16	-6	-26	-27

Table 5.25 Max-Latency *GAIN*

AI/ms	Class I %	Class II %			Class III %
		25ms	50ms	100ms	
5	-3	-14	-18	-22	-32
20	4	-16	-20	-26	-38
38	8	-17	-12	-23	-38
75	0	-16	-9	-27	-47
150	4	-3	-3	-29	-40

Table 5.26 Min-Latency *GAIN*

5.6 Bandwidth consumption of WAN

In this section, we will show that protocol *[Mencius]*^{N} has lower bandwidth consumption than *Mencius*. Protocol *[Mencius]*^{N} reduces the number of exchanged messages over wide-area network between sites to finalize each instance. This approach is behind the reduction in bandwidth consumption by *[Mencius]*^{N}.

This section is explained in two sections: first, this is issue is explained in failure-free situation and second, under failure when there is a need for revocation.

5.6.1 Bandwidth consumption in failure-free situation

In using *Mencius* on a system of three sites to generate three consecutive orders, each site needs to run one instance of Paxos. Paxos instance in failure-free situation uses three types of messages (*SUGGEST, ACCEPT,* and *LEARN*). The total number of messages exchanged over wide-area network needed to generate three consecutive orders is 9 messages. Nevertheless, using protocol *[Mencius]*^{N}, the total number of messages exchanged over wide-area network needed to report three consecutive orders is 3 messages only. It is known that in protocol *[Mencius]*^{N}, Paxos is executed on local level, but at global level, only *LEARN* message is sent to inform other sites about the outcome of each instance. Protocol *[Mencius]*^{N} consumes 3/9 of the bandwidth that is consumed by *Mencius.*

5.6.2 Bandwidth consumption under failure

It is assumed that *Mencius* has three sites and one of them crashes. One of the other two correct sites starts revocation. In this case, 2 instances will execute Paxos in failure-free situation which will use three messages (*SUGGEST, ACCEPT,* and *LEARN*) and the third instance will go through revocation will be executing instance of Paxos with five messages (*PREPARE, ACK, SUGGEST, ACCEPT,* and *LEARN*). The total number of messages exchanged over wide-area network needed to generate three consecutive orders is 6 + 5 = 11 messages. Again using protocol *[Mencius]*^{N}, the total number of messages exchanged over wide-area network needed to report three consecutive orders is 3 only. Protocol *[Mencius]*^{N} consumes 3/11 of the bandwidth that is consumed by *Mencius.* What is presented here regarding *Mencius* is the best scenario in case of failure, which means one site crashes and only another correct one succeed in revoking it. Nevertheless, there are cases in which more than

one correct server competes to revoke the crashed one, which means more messages are needed in revocation. The number of messages may go to infinity.

Bandwidth consumption is improved in protocol *[Mencius]*^{N} compared to *Mencius* to *1/3* in failure-free situation, *1/3.67* under failure and only one correct server succeed in revoking the crashed one, or even $1/\infty$ when there is a competition for revocation.

5.7 Removing client blocking

Mencius suffers from client blocking, which means that clients do not make progress while their server is crashed. Our protocol overcomes this issue by providing each site with more machines. Unlike *Mencius*, in protocol *[Mencius]*^{N}, the service on each site is replicated, each site with *3* nodes. Hence, as long as the majority is correct in each site, the clients will never be blocked. This will guarantee that clients connected to that site will be served, progress is made and the site will be able to order requests received from its group of clients.

5.8 Performance Assessment of Revised *Mencius*

In chapter 3 of this thesis, we presented another version of multi-ordering protocol synonym to *Mencius* which is called Revised *Mencius*. In that chapter, the issue of revocation is addressed and a new version of *Mencius* that reduces the overhead of revocation is presented. The main point of concern, instead of revoking instance by instance, a whole range using one instance only is revoked. The design is built on this concept and in the following sections, results are presented.

In failure-free situations, both *Mencius* and our revised version have almost the same performance regarding latency and throughput. In our experiments, failure-free situation is omitted and performance analysis is focused on comparing both protocols during revocation process only.

A number of experiments to compare both versions regarding their performance in case of revocation are run. The comparison is carried out using a three sites system; one site is forced to be faulty and one of the other two correct sites starts revocation**.** The results of the experiments are divided into two sets, each having two tables and one graph. One set represents throughput and the other set represents latency. These

results could be found in Table B.21 to Table B.24 and in Figures B.11 and Figure B.12 in appendix B at the end of this thesis.

5.8.1 Throughput

To compare throughput, Long Message is used which has already been defined. A number of experiments that has 100ms time delay of the virtual link enforced by DummyNet is used. Each experiment has its own time interval between requests generated by clients. The following are the time interval values chosen from uniform distribution (10ms, 20ms, 38ms, 75ms, 150ms, 1000ms and 10000ms).

Testing both protocols under low rate of requests such as 75ms, 150ms, 1000ms and 10000ms shows no difference regarding throughput. Under these rates there is no demand for processing power, thus underutilizing the processing capacity of the CPU's of the correct servers.

On the other hand, when rates are increased to 38ms, 20ms and 10ms, significant differences in performance are observed, especially with 10ms. The concept of revocation in *Mencius* is different to the one in the revised version of *Mencius*, which is believed to be the reason behind the production of different throughput for both protocols.

In *Mencius*, with the increase of load, there is more demand for processing power especially on the side of the server responsible for revocation. This is because in addition to ordering its own proposals, it is in charge of generating and ordering proposals on behalf of the suspected or crashed server. More processing power is needed from the revoker, which reduces its throughput. While in the revised version of *Mencius* during revocation, both correct servers suggest their own proposals only, as revocation of the whole range is carried out using one instance only. In such case, both correct servers have enough processing capacity which enables them to cope with higher loads. The summary of the results are shown in Table 5.27.

AI ms	**10**	**20**	**38**	**75**	**150**	**1000**	**10000**
Class II 100ms	24	14	7	0	0	0	0

Table 5.27 throughput Gain

5.8.2 Latency

Theoretically, it can be calculated that the latency of both protocols are as follows: latency for *Mencius* for three consecutive instances is the total of two instances of three messages and one instance of five messages (300+300+500=1100/3=367ms). Latency for revised version of *Mencius* is the total of two instances of three messages only (300+300=600/2=300ms). To evaluate latency, two latency values, max-Latency and min-Latency which have already been defined, are distinguished.

We find that the difference between the two protocols under low load is around 60ms, which is close to the value calculated above, indicates that revised version of *Mencius* has better performance. This difference, however, goes up as the load is increased. For example, at the highest load, the difference is around 100ms, again in favour of the revised version of *Mencius*.

A significant benefit is gained from the modification applied to revocation. In revised versions of *Mencius*, the revoker uses one instance only to revoke a whole range. After that, it will continue its normal job in ordering its own proposals. This will make the revoker and the other correct servers work with the same processing capacity. From a latency perspective in such circumstances, revised version of *Mencius* has better performance than *Mencius*. The summary of results is shown in Table 5.28 and Table 5.29.

AI ms	**10**	**20**	**38**	**75**	**150**	**1000**	**10000**
Class II 100ms	-20	-18	-16	-16	-16	-16	-16

Table 5.28 Max-Latency *GAIN*

AI ms	**10**	**20**	**38**	**75**	**150**	**1000**	**10000**
Class II 100ms	-23	-18	-16	-19	-19	-20	-19

Table 5.29 Min-Latency *GAIN*

5.9 Summary

A significant benefit of building the new protocol on top of *Mencius* abstraction and Paxos abstraction is that the solution ensures liveness and safety. In addition, it was proven that the objectives of the new protocol presented in this thesis are fulfilled. The concept of using *Mencius* on local-area network provides us with great benefits regarding client progress, bandwidth consumption and latency.

Clients make progress as long as the majority of servers are correct at site level and bandwidth consumption is reduced to at least 1/3 of *Mencius.* Latency is reduced especially when the time delay of the virtual link goes up. Threshold of saturation is also higher.

The results of the experiments revealed that there is a trade-off between throughput and latency in relation to message size. If the size of our request is short, then the following is suggested: when the request rate is low, single messages (no batching) is used. This produces better latency. However, if the rate is high, users should resort to batching multiple messages in a single message in order to get better throughput.

Mencius was not compared with $[Mencius]^N$ regarding revocation. It is known that revocation is an execution of instance of Paxos and in the new protocol at global level, instances of Paxos are not executed, meaning revocation does not exist at global level. This explains why such a comparison was not carried out. Although a comparison between *Mencius* and revised version of *Mencius* is made, the structure of revised version of *Mencius* is found in chapter 3 and the results of comparison can be found in section 5.9 of this chapter.

The main conclusions: It is advised that those who want to solve *Total Order* problem over local-area network use either *Mencius* or revised *Mencius,* as these two protocols overweight $[Mencius]^N$. However, for those who want to solve it over wide-area network, its preferable to use $[Mencius]^N$, which is proven throughout the course of this thesis to have higher performance.

Chapter 6

Summary and conclusion

This thesis introduced a new class of High-Performance Multi-Ordering Protocol that optimally uses *Mencius* protocol on top of Paxos to circumvent the FLP impossibility. It presents the design and the analysis of our protocol $[Mencius]^N$.

In chapter 3, an alternative protocol to *Mencius* was introduced which has the same system context regarding the number of servers and sites but has a different set of assumptions. The revised version of *Mencius* in chapter 3 criticises *Mencius* from revocation perspective under a new assumption. In the second part of this work, we presented our main protocol $[Mencius]^N$ that tackles many issues raised by *Mencius*.

The performance of our proposed protocol $[Mencius]^N$ was extensively examined and compared to the *Mencius* in various settings. The results received from this are encouraging. We will summarize the work presented in our thesis and present directions for future work.

6.1 Summary

Total Order problem is a well-known issue at the core of building dependable distributed systems. Particularly in state machine replication technique, where replicas need to agree on various issues such as order of client request execution, group membership, transaction commitment, etc., reaching consensus is fundamental to solve any of these agreement problems. At the same time, it has been proven that finding a deterministic solution cannot be guaranteed in asynchronous network setups where replicas are fault-prone. Circumventing FLP impossibility has been an active research area in past two decades. The most common approaches that have been proposed can be categorized into four types; randomised protocols, fail-signal protocols, deterministic protocols, and multi-ordering protocols.

Randomized protocols [EMR01, MNC+06] are a family of protocols where FLP result is avoided by providing a probabilistic solution. Participants go over rounds of communication and make random choices on their estimate of decision values. The protocol progresses in such a way that eventually an identical value is decided. These protocols guarantee termination only in probabilistic terms which tends to 1 as

elapsed time approaches infinity. This type of protocol is a non-leader protocol; however, the main disadvantage of this protocol is that the number of messages needed for termination is unknown and the time needed to arrive at a decision may approach infinity.

The next one is called Fail-Signal protocol. Fail-Signal [BES+96, IE06] protocol is the third in the family of inherently redundant processes; namely, fail-stop and fail-silent processes and is constructed in a similar way. FS is a protocol whose termination guarantee is not dependant on any systemic/network conditions and the performance is only affected by existing communication delays and real failures. Fail-Signal process circumvents the impossibility by making the failing process announce its imminent failure and stop working after failing. The main advantage of this type of protocol is the use of perfect failure detector. However, the main disadvantages of this approach is that each FS node consists of at least two machines connected by a synchronous network. This will result in a higher level of message complexity because all constituents of FS node will generate their own messages. For example, if FS node has two machines, then 2 identical proposals will be sent out to all correct FS nodes. Two identical *ack* messages will be sent out to the other FS node and 2 identical Learn messages will be sent out to all correct FS nodes (message redundancy). Another disadvantage is the high latency that results from waiting for the response from all processes.

The third type, Deterministic protocols, is built on the concept of Unreliable Failure Detector [CT91, CT96, CHT96]. Each process accesses *Failure-Detector oracle*, which provides a list of processes suspected to be crashed. The weakest form of Failure-Detector is denoted by $\Diamond S$, which allows it to solve consensus. This type of FD has the following properties: (1) any crashed process is eventually suspected (completeness), (2) there is a time after which correct processes are not suspected (eventual weak accuracy). This category of protocols tends to be coordinator-based. A specific process is given the role of coordination of the execution of the protocol. When it is crashed, the protocol then chooses another process to play this role. Paxos [LAM98] and Chandra Toueg [CT91] are considered to be the pioneers in this group of protocols. Compared to the other two categories, deterministic approach is characterized by its lower latency and lower level of message complexity as well.

The last type is multi-ordering protocols. We consider *Mencius* [MJM08] as a novel and new protocol belonging to this group. *Mencius* is a replicated state machine

built on the abstraction of Paxos which runs concurrent instances of Paxos. *Mencius* as a multi-ordering protocol tackles the issue of single leader bottleneck inherited from Paxos. Paxos suffers from some drawbacks in terms of communication pattern, CPU processing capacity, and latency of learning the outcome. By tackling the problem of single leader, the throughput is increased under high client load and latency is lowered under low client load

The protocol presented in this thesis belongs to the last group. It is built on top of *Mencius* [MJM08] which, in turn, is built on top of Paxos protocol [LAM01, LAMO6]. Paxos forms the underlying protocol and the core protocol used to circumvent FLP impossibility. This work consists of two parts. In chapter 3, the first part is presented in which the issue of revocation in *Mencius* is addressed. The construction of *Mencius*, in particular, is based on a claim which says false suspicion and crash rarely occur. This work proposes that cases of false suspicion and crash occur frequently and the cost of revocation is very high. To minimize that cost, certain changes to *Mencius* protocol were made. The main modification is that a whole range using one instance is revoked, instead of being revoked one by one.

The new revised version of *Mencius* presented in chapter 3 addresses the problem of the high probability of crash and false suspicion that might trigger more frequent FD. The structure followed in presenting the revised version of *Mencius* is the same as the one adopted in *Mencius.* The assumptions and principles are explained in terms of rules and optimizations; the main differences between the two protocols lay in how revocation is carried out. In the revised *Mencius*, as soon as one server starts to revoke a suspected server, it will try to revoke all instances in a certain range. The process of revocation will start from the smallest instance that should have been coordinated by the suspected server. If the revocation of that instance was successful, then the whole range will be revoked automatically; otherwise the revoker will receive at least one *NACK* which means revocation will be aborted. We proved that the correctness of the new version is inherited from the old version of *Mencius.* All properties of consensus protocol are fulfilled, hence safety and liveness are ensured.

In the forth chapter of the thesis, the main work is presented where a new protocol, called $[Mencius]^N$, is proposed. The challenges of the new protocol are how to address issues such as latency, throughput, threshold of saturation, client blocking and bandwidth consumption. The aim of the work is to develop a protocol that fulfils the objectives by reducing latency to client requests, increasing threshold of

saturation, guaranteeing that clients make progress even in case of crash and lastly, reducing bandwidth consumption.

The proposed solution is to build multiple cooperative *Mencius'* as a two-layer system. One layer consists of local *Mencius* and the second layer forms global multi-ordering protocol. The underlying network connecting servers of each local *Mencius* system will be based on FIFO communication channels (LAN), however, the underlying network connecting global *Mencius* systems will be based on asynchronous communication channels (WAN). The new system consists of two replicas, global replica and local replica. Global replica exists on a level of sites implementing a replicated state machine. Each site represents one abstraction of *Mencius*, *N* sites (where $N \geq 2$) creating $[Mencius]^N$. Local replica (site) is on the level of servers where each local replica consists of n servers $(n = 2f + 1)$, thus forming a local *Mencius*.

We consider each site as a coordinator on a global level, ordering requests received from its own group of clients. Each site has one distinguished server which will be in charge of talking to other sites. This server is called *Site Speaker*. Communication between sites only takes place between *Site Speakers*, so *Site Speaker* is a normal server that has more jobs to do than other servers in the system.

Each site using its own local *Mencius* will produce a local stream of instances. Streams produced locally will be exchanged by *Site Speakers* and merged by all servers to form global stream or $[Mencius]^N$ stream.

The replacement of crashed or suspected *Site Speaker* server in $[Mencius]^N$ servers in each site are divided into two groups according to their function. One group consists of one server which is called *Site Speaker*. The second group consists of the remainder of the servers which are called the normal group. Suspicion of failure or crash in the normal group is treated in terms of revocation only, exactly as in *Mencius*. This process guarantees that local *Mencius* can make progress. Nevertheless, suspicion of failure or crash of the *Site Speaker* should be solved in two directions: one direction is revocation as the other group, and the second direction is to replace the *Site Speaker*. Protocol $[Mencius]^N$ relies on *Site Speakers* communication to make progress. This server is prone to crash, false suspicion or overload of work. These problems can be classified according to the reaction taken by the system into two groups: one group consisting of crash and false suspicion, the second group consisting of overload. When the current *Site Speaker* suffers from

crash or false suspicion, then the system will enforce a new *Site Speaker*, which means the correct servers will choose a new *Site Speaker* to replace the current one. Nevertheless, when the current *Site Speaker* is overloaded, it then asks for a replacement. This is done for the sake of load distribution balance.

All servers are publicly ranked and therefore defined by the sequence in which they are to be the *Site Speaker.* At initialization time, server s_1 will be the *Site Speaker* at all sites and as soon as s_1 is suspected or crashed, then the next server in ranking will take over as the new *Site Speaker*. This choice is based on rotating coordinator paradigm in a round-robin fashion.

The other method of replacing the *Site Speaker* is done for the sake of load distribution balance. In addition to coordinating its client requests, *Site Speaker* has to send local site stream to all other sites, receive streams from other sites and multicast them to local servers. This reflects the fact that network traffic from or to the *Site Speaker* is much higher than its counterparts in the site. The second issue is that as *Site Speaker* receives higher number of messages, there is a higher demand for processing power than other servers. In trying to get a more balanced communication pattern and better CPU system utilization, the current *Site Speaker* will seek the server with the lowest load and hand to it the *Site Speaker* responsibility. Individual streams from each server will be received by all correct servers in that site. This will enable the *Site Speaker* to calculate the load distribution of each server and find out which server has the highest load and which one has the lowest load. The *Site Speaker* will choose the server with the lowest load and ask it to be the new *Site Speaker.*

As soon as some server successfully installs itself as the new *Site Speaker,* it starts communication with other sites by multicasting learn messages to all servers, *1 to n,* which will inform all servers at other sites about *Site Speaker* change. The process of multicasting *1 to n* will continue until the new *Site Speaker* receives learn message from other *Site Speakers.* It will then switch from *1 to n* communication to *1 to 1*. The main reason for *1 to n* communication, started by the new *Site Speaker*, is to eliminate the deadlock that might be created by the crash of more than one *Site Speaker* at the same time.

Finally Chapter 5 is dedicated to the results of our experiments and their analysis. It can be proved that the objectives of the new protocol presented in this thesis are fulfilled.

6.2 Conclusion

We developed a protocol belonging to a family of Crash fault-tolerant order protocols by exploiting *Mencius* abstraction. Only crash model failure is investigated within this thesis. We are able to achieve our objectives by building our protocol on the concept of multiple cooperative *Mencius*'. Clients make progress as long as the majority of servers are correct at site level and bandwidth consumption is reduced to at least 1/3 of *Mencius.* Latency is reduced especially when the time delay of the virtual link went up. The threshold of saturation is also higher. These benefits come at a cost of extra machines at each site. In *Mencius,* each site has one machine while in $[Mencius]^N$, the minimum number of machines at each site is *n* machines $(n=2f + 1)$. The cost is not considered very high as hardware prices are going down.

Our main conclusions can be summarized on the following points:

According to the environment and network topology one can make a trade-off between *Mencius* and $[Mencius]^N$, we can consider the following scenarios:

1- All sites without any cluster, assuming that all sites connected through WAN. The implementation of *Mencius* or $[Mencius]^N$ in such an environment depends on the number of sites:

- If the number of sites $n < 6$ we can use only *Mencius.* Because in case of $n = 5$ we cannot form $[Mencius]^N$ of $N = 2$.
- However, if the number of sites $n \geq 6$ we can use both protocols. For example if n = 9 it is preferable to use $[Mencius]^N$ rather than *Mencius* to solve *Total Order* problem. In such case both protocols have the following facts:
  (1) The implementation of *Mencius* will produce a majority of 5 which will increase latency. However for $[Mencius]^N$ the majority at local level is not changed.
  (2) In *Mencius* the number of messages needed to finalize each instance is $3(n-1) = 24$ *msg*, however for $[Mencius]^N$ the maximum number of messages needed to finalize each instance is 6 msg at local level plus 6 msg at global level which will produce a total of 12 msg.

for solving *Total Order* problem over wide-area network, it is preferable to use $[Mencius]^N$, because it has higher performance.

2- All sites or servers are kept within one cluster, in such scenario our advice is to use either *Mencius* or revised *Mencius* to solve *Total Order* problem over local-area network, as these two protocols overweight $[Mencius]^N$.

3- Some sites without any cluster, this type of scenario is not investigated in this work. Also in this case we can conclude as in point 1 the redundancy level makes the difference:

- If the number of sites $n < 6$ we can use only *Mencius.* Because in case of $n = 5$ we cannot form $[Mencius]^N$ of $N = 2$.
- However, if the number of sites $n \geq 6$ both protocols can be used. From our experience we can say that protocols will suffer from higher latency, this comes as a result of implementing some nodes over wide area network. Nevertheless to find out exactly which protocol to use in such scenario we need to make more experiments.

4- Exploiting message size to increase performance in $[Mencius]^N$, we find that there is a trade-off between throughput and latency in relation to message size. If the size of the request is short, then the following is suggested: when the request rate is low, single messages (no batching) is used. This produces better latency. However, if the rate is high, users should resort to batching multiple messages in a single message in order to get better throughput.

The major drawback of $[Mencius]^N$ is its high level of redundancy in comparison with *Mencius*. For example, in this work, the level of redundancy was three times higher. Each machine in *Mencius* is equivalent to three machines in $[Mencius]^N$. This negative aspect can be exploited and transferred to a positive one. This can be achieved by deploying different distributed application services on $[Mencius]^N$. For example, in this work, three different services can be deployed. The suggested solution to the problem of high level of redundancy in terms of extra hardware will reduce its cost.

In the following scenario of three distributed application services needing to be deployed on wide-area network, there are two choices in solving this problem:

1- Either three independent *Mencius;* each having three servers are used, which forms a total of nine machines.

2- Or, we resort to *[Mencius]*^{3}, each site having three machines forming a total of nine machines as well.

The above mentioned scenario illustrates that the cost of hardware in both cases is the same, but using *Mencius* implies the inheritance of all the flaws of the system as well. However, using *[Mencius]*^{N} avoids these problems as proven during the course of this work.

6.3 Future work

Through the course of this thesis, we find that *Total-Order* problem is solved over local-area network. In line with that, our future work will concentrate on improving the performance of that part of the protocol executed over local-area network. We suggest two future works:

6.3.1

First, Revised Mencius can be improved by introducing two modifications:

1. Implementing Mencius on local area network using UDP protocol over IP multicast services instead of TCP protocol.
2. We suggest fourth optimization to replace the accelerator. This point will be explained according to the following system context; 3 servers *p, q,* and *r*. Server *p* is active while the other two servers *q* and *r* are idles. No *SKIP* message will be explicitly sent separately between idle servers (no accelerator), such as *q* and *r* in our scenario. Servers *q* and *r* send *ACCEPT* as a response for *SUGGEST* received from *p* for instance *i*, implies that they are *SKIPing* their turn. Server *p* piggybacks *SKIP* messages received from *q* and *r* on *LEARN* for instance *i.* After receiving *LEARN* message for instance *i*, server *q* will learn about server *r* status and vice versa.

Using the aforementioned optimization will eliminate the need of accelerator and the blocking of the two idle servers will be removed. As the waiting time for the accelerator to be triggered is eliminated, overall *Mencius* latency will benefit.

6.3.2

For a second future work, we propose a Multi-Ordering Protocol based on *Randomized* Consensus Protocol. *Randomized* protocols are non-leader protocols; all

nodes in such protocols have the same quality and the same responsibility and have no use of unreliable Failure Detectors. One of the features of such protocols is that all communications in such protocol goes from *1 to n*, so we propose here to implement *Randomized* Multi-Ordering Protocol on local-area network using UDP protocol over IP multicast services as the underlying protocol instead of TCP protocol. This will reduce the number of times needed by the application layer to send each message to all nodes. Only one action taken by IP multicast services to deliver each message to all nodes registered with that group. This protocol will be implemented on local-area network to solve *Total Order* problem. Finally, performance comparison will be carried out between *Randomized* protocol with *Mencius* protocol. This comparison will reveal the potential of *Randomized* protocol and give us a clear view about *Randomized* protocol as a Multi-ordering protocol used to solve *Total Order* problem over local-area network.

Appendix A

A.1 Evaluation for short messages

A.1.1 Throughput

[Mencius]^N	
Arrival Intervals/ms	Throughput
5	465
20	140
38	76
75	39
150	20

Table A.1: throughput *[Mencius]*^{N}

Mencius	
Arrival Intervals/ms	Throughput
5	333
20	124
38	71
75	38
150	19

Table A.2: throughput *Mencius*

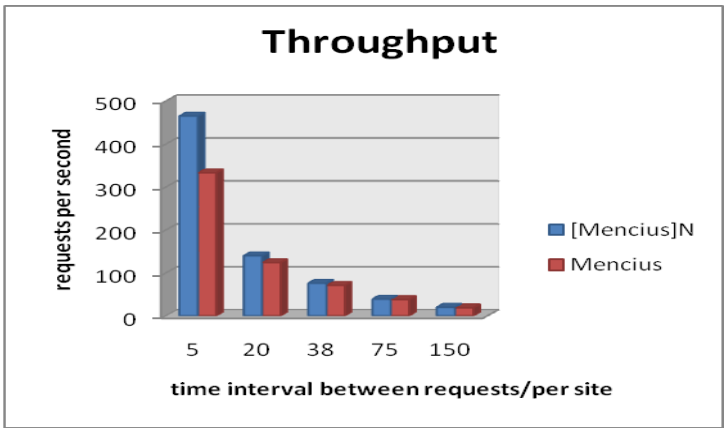

Figure A.1: Experiment 1; time delay of LAN, no DummyNet.

[Mencius]^N -25	
Arrival Intervals/ms	Throughput
5	425
20	139
38	75
75	39
150	20

Table A.3: throughput *[Mencius]*^{N} -25

Mencius -25	
Arrival Intervals/ms	Throughput
5	332
20	124
38	71
75	38
150	19

Table A.4: throughput *Mencius* -25

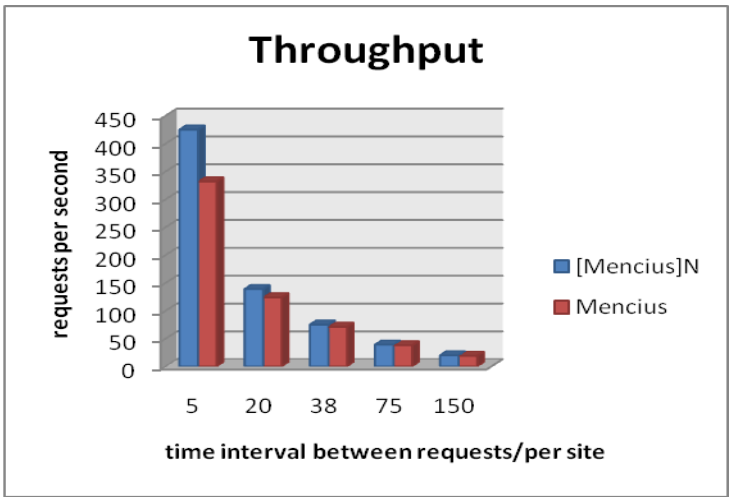

Figure A.2: Experiment 2; one-way delay of 25ms.

[Mencius] ^{N} -50	
Arrival Intervals/ms	throughput
5	465
20	140
38	76
75	39
150	20

Table A.5: throughput *[Mencius]* ^{N} **–50**

Mencius -50	
Arrival Intervals/ms	throughput
5	329
20	124
38	71
75	38
150	19

Table A.6: throughput *Mencius* –50

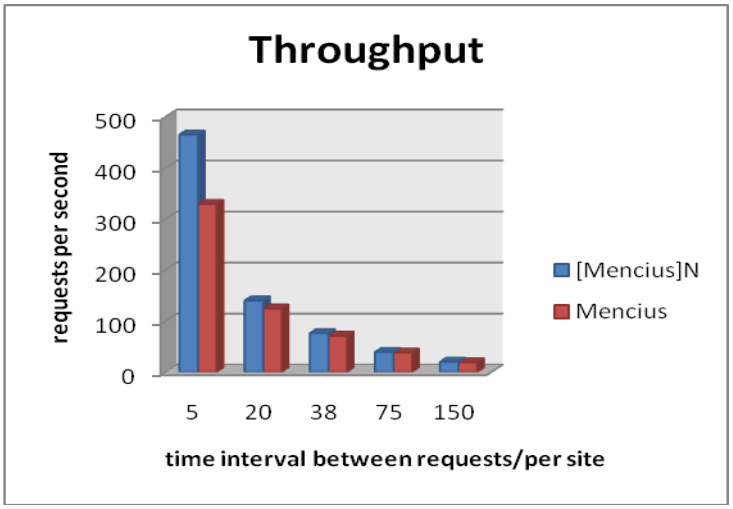

Figure A.3: Experiment 3; one-way delay of 50ms.

[Mencius]^N -100	
Arrival Intervals/ms	Throughput
5	430
20	138
38	75
75	39
150	20

Table A.7: throughput *[Mencius]*^{N} –100

Mencius -100	
Arrival Intervals/ms	Throughput
5	331
20	124
38	71
75	38
150	19

Table A.8: throughput *Mencius* –100

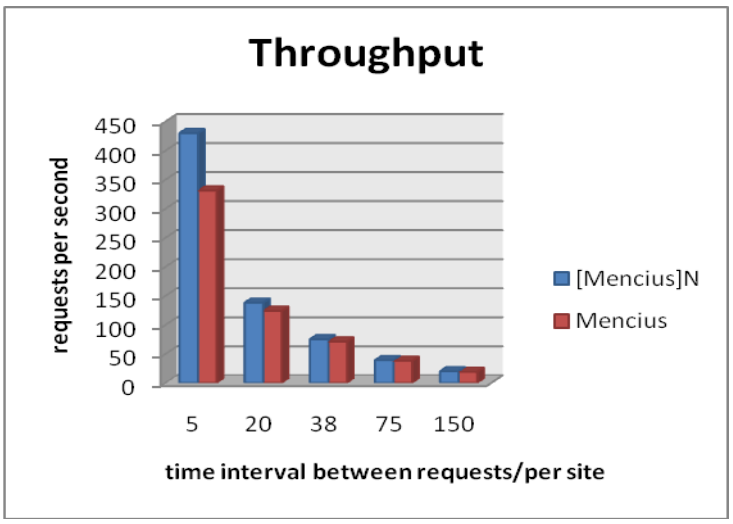

Figure A.4: Experiment 4; one-way delay of 100ms.

[Mencius]^N -D	
Arrival Intervals/ms	Throughput
5	411
20	135
38	74
75	39
150	20

Table A.9: throughput *[Mencius]*^{N}-D

Mencius -D	
Arrival Intervals/ms	Throughput
5	321
20	122
38	71
75	38
150	19

Table A.10: throughput *Mencius*-D

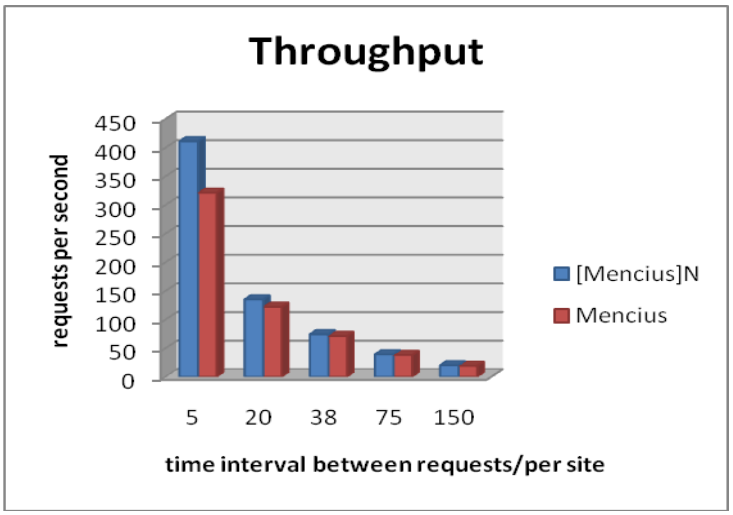

Figure A.5: Experiment 5; random one-way time delays.

A.1.2 Latency

[Mencius]^N		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	406.44	309.78
20	414.56	302.1
38	412.68	310.77
75	413.22	317.8
150	422.1	311.97

Table A.11: latency *[Mencius]*^{N}

Mencius		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	412.4	320.56
20	410.64	290.1
38	409.9	287.35
75	416.23	318.27
150	407	298.89

Table A.12: latency *Mencius*

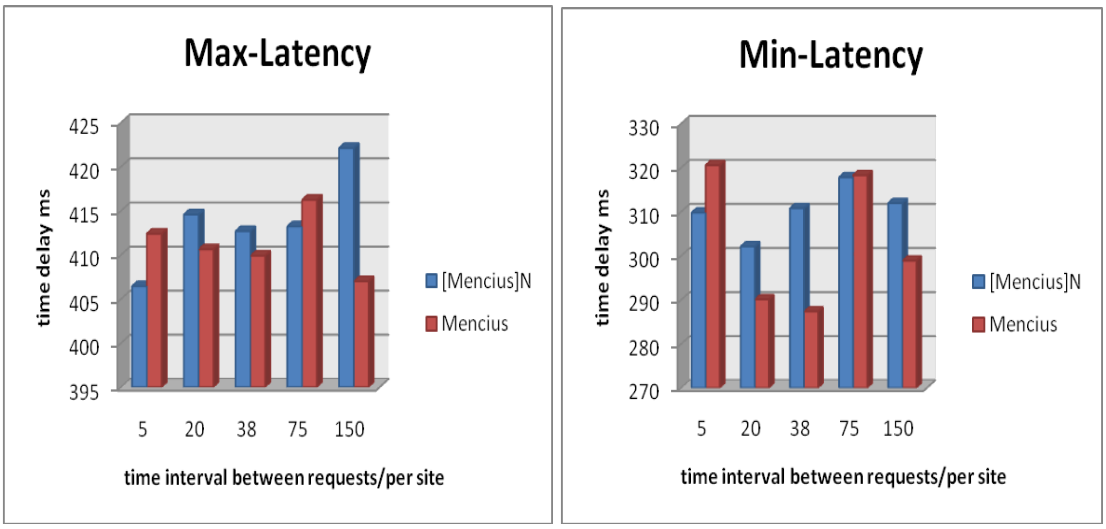

Figure A.6: Experiment 1; time delay of LAN, no DummyNet.

[Mencius]^N -25		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	418.5	314.4
20	416.6	315
38	414	313.5
75	410.9	314.8
150	404.4	309.9

Table A.13: latency *[Mencius]*^{N} D25

Mencius -25		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	444.5	366.39
20	450	374.3
38	468.3	378
75	479.53	375.76
150	480.23	319.15

Table A.14: latency *Mencius* D25

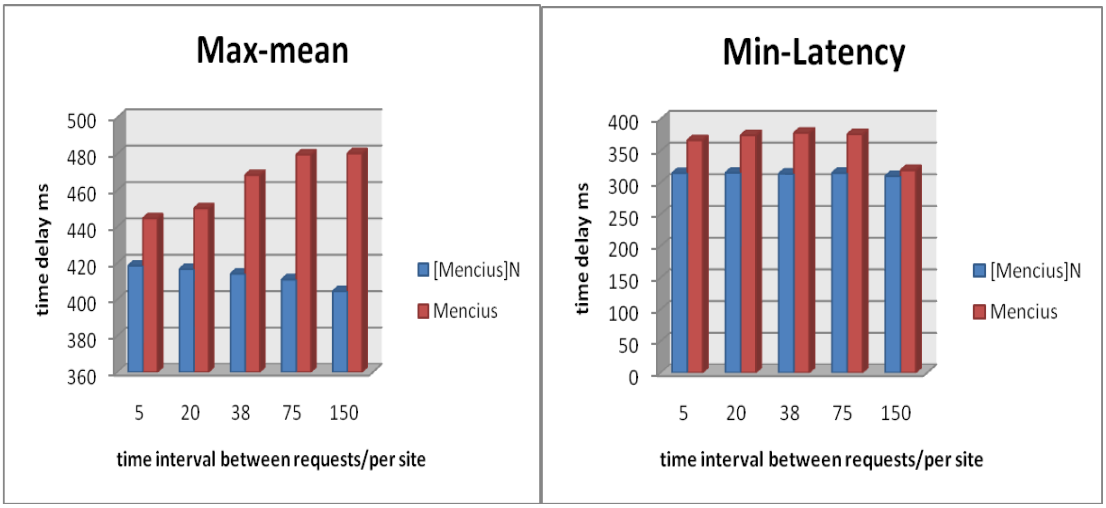

Figure A.7: Experiment 2; one-way delay of 25ms.

[Mencius]^N -50		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	426.36	343.8
20	441.8	320.6
38	449.5	337.9
75	459.7	339.1
150	480.1	360.2

Table A.15: latency *[Mencius]*^{N} -50

Mencius -50		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	490.16	416.9
20	498.06	402.38
38	485.4	384.45
75	500	371.4
150	509.07	373.14

Table A.16: latency *Mencius*-50

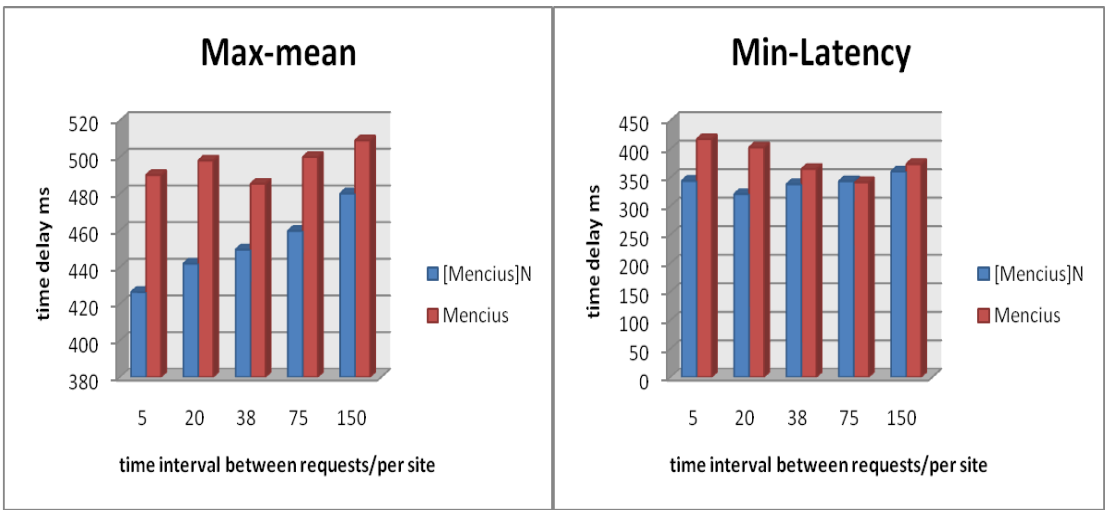

Figure A.8: Experiment 3; one-way delay of 50ms

[Mencius]^N -100		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	496.6	391.8
20	481.3	382.6
38	481.9	395.7
75	488.2	403.5
150	494.7	398

Table A.17: latency *[Mencius]*^{N} 100

Mencius -100		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	641.27	504.8
20	597.37	514.59
38	661.64	515.94
75	651.83	549.54
150	668.83	560.76

Table A.18: latency *Mencius* 100

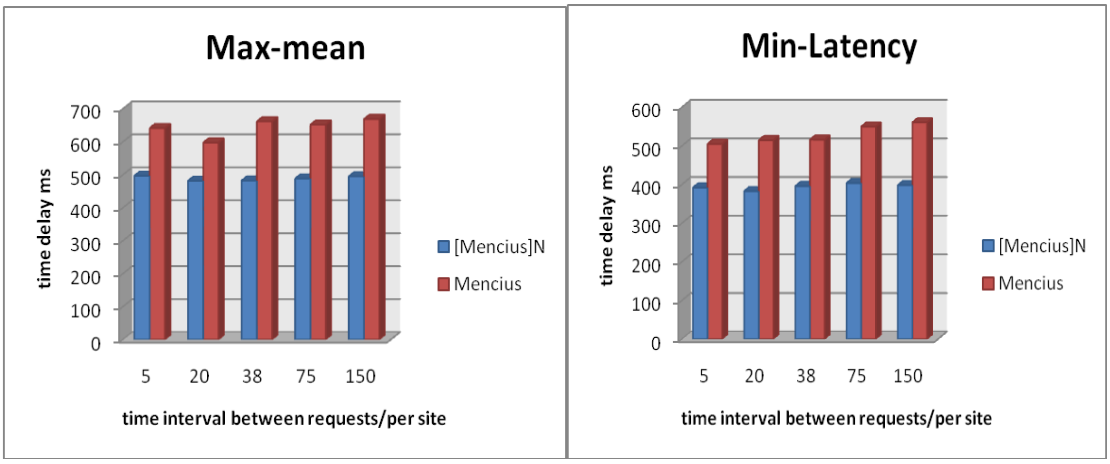

Figure A.9: Experiment 4; one-way delay of 100ms.

[Mencius]^N -D		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	1946	1203
20	1884	1089
38	1703	1059
75	1638	928
150	1506	998

Table A.19: latency *[Mencius]*^{N} D

Mencius -D		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
5	2415	1781
20	2190	1748
38	2177	1697
75	2153	1738
150	2071	1667

Table A.20: latency *Mencius* D

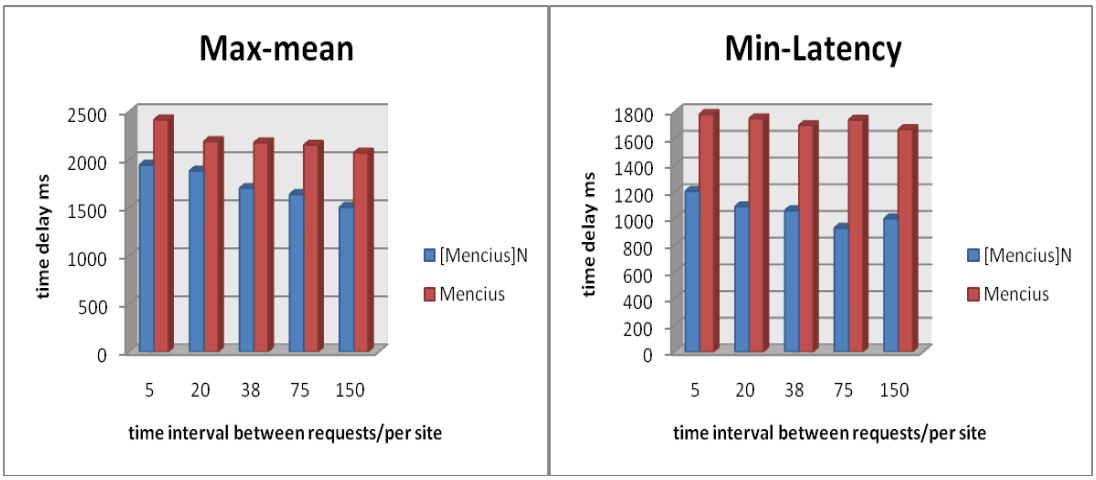

Figure A.10: Experiment 5; random one-way time delays.

A.1.3 Performance Assessment of Revised *Mencius*

***Mencius*-100**	
Arrival Intervals/ms	Throughput
10	155
20	85
38	45
75	26
150	13
1000	2
10000	0.2

Table A.21: throughput *Mencius*

Revised *Mencius*-100	
Arrival Intervals/ms	Throughput
10	192
20	97
38	48
75	26
150	13
1000	2
10000	0.2

Table A.22: throughput revised *Mencius*

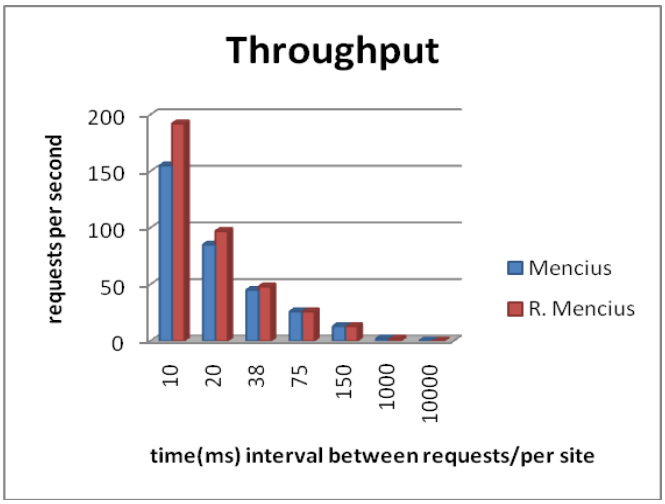

Figure A.11: One-way delay of 100ms.

***Mencius*-100**		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
10	510.71	451.77
20	478.41	391.47
38	465.28	408.34
75	462.83	379.4
150	439.1	357.66
1000	395.75	314.23
10000	385.69	322.59

Table A.23: latency of *Mencius*

Revised *Mencius*-100		
Arrival Intervals/ms	max-Latency/ms	min-Latency/ms
10	407.21	348.27
20	392.21	321.37
38	390.52	342.34
75	388.13	306.7
150	370.39	290.96
1000	331.05	252.53
10000	324.99	262.89

Table A.24: latency of *revised Mencius*

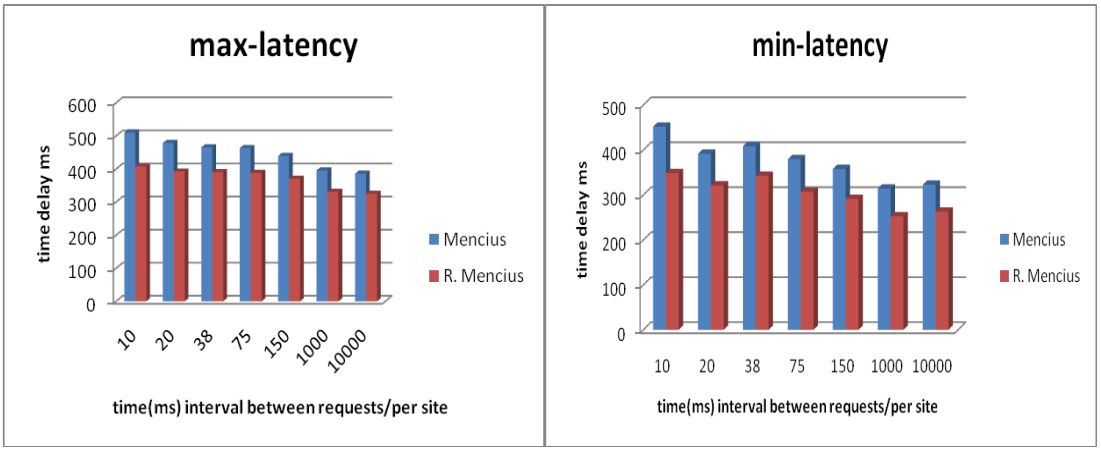

Figure A.12: One-way delay of 100ms.

References

[ABA12] Daniel J. Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design”. Computer: Innovative Technology for Computer Professionals, IEEE Computer Society, pages 37-42, February 2012.

[AE11] K. Alekeish and P. Ezhilchelvan, “Consensus in Sparse, Mobile Ad-hoc Networks,” May 2011.

[AH90] J. Aspnes and M. Herlihy. “Fast Randomized Consensus Using Shared Memory”, *Journal of Algorithms,* vol. 11, no. 3, pp. 441-460, 1990.

[ASP03] J. Aspnes, “Randomized Protocols for Asynchronous Consensus”, *Distributed Computing*, vol. 16, no. 2-3:165-175, September 2003.

[BAU05] Baumann, R., “Soft Errors in Advanced Computer Systems”, *IEEE Design and Test of Computers*, pp. 258-266, May-June 2005.

[BEN83] M. Ben-Or. “Another advantage of free choice: completely asynchronous agreement protocols”. *In Proceedings of 2nd ACM Symposium on Principles of Distributed Computing*, pp. 27-30, 1983.

[BES+96] F.V. Brasileiro, P.D. Ezhilchelvan, and S.K. Shrivastava, N.A. Speirs, S. Tao, “Implementing Fail-Silent Nodes for Distributed Systems”, *IEEE Transactions on Computers*, 45(11): pp 1226-1238, 1996.

[BM92] N. Budhijara, K. Marzullo “Tradeoffs in Implementing Primary-Backup Protocols”, Technical Report TR 92-1307, Department of Computer Science, Cornell University, 1992.

[BMST93] N. Budhijara, K. Marzullo, F. Schneider, and S. Toueg “The Primary Backup approach”, In S. Mullender, (ed.), Distributed Systems, pp. 199-216. Wokingham: Addison-Wesly, 2nd ed., 1993.

[BOI01] R. Boichat. “Reliable and Total Order Broadcast in the Crash Recovery Model”. PhD thesis, ´Ecole Polytechnique Federale Lausanne, Switzerland, Nov. 2001.

[BRE12] E. Brewer, “CAP Twelve Years Later: How the “Rules” Have Changed”. Computer: Innovative Technology for Computer Professionals, IEEE Computer Society, pages 23-29, February 2012.

[CD89] B. Chor, and C. Dwork, “Randomization in Byzantine agreement”. *Advan. Comput. Res*. 5, 443-497, 1989

[CHT96] T. Chandra, V. Hadzilacos, and S. Toueg. “The weakest failure detector for solving consensus”. Journal of the ACM, 43(4):685–722, 1996.

[CMS89] M. Chor, M. Merritt, and D.B. Shmoys, “Simple Constant-Time Consensus Protocols in Realistic Failure Models”, *Journal of the ACM*, 36(3): 591–614, 1989.

[CR09] M. Carbone, L. Rizzo, Dummynet Revisited. Department of Information Engineering, Pisa University. November 2009.

[CR+09] Y. Chen[1], A.Romanovsky[2], (A. Gorbenko, V. Kharchenko, S. Mamutov, O. Tarasyuk)[3]. “Benchmarking Dependability of a System Biology Application”. [1]Institute for Ageing and Health [2]School of Computing Science Newcastle University Newcastle-upon-Tyne, UK. [3]Department of Computer Systems and Networks National Aerospace University Kharkiv, Ukraine.

[CS00] B. Charron-Bost, A. Schiper, “Uniform Consensus is Harder Than Consensus (extended abstract)”, Technical Report LSR-REPORT-2000-014, Ecole Polytechnique Federale de Lausanne, Switzerland, 2000.

[CT91] T.D. Chandra and S. Toueg, “Unreliable failure detectors for asynchronous systems (Preliminary version)”. *In Proceedings of the tenth annual ACM symposium*

on Principles of distributed computing, pp. 325 – 340, Montreal, Quebec, Canada, 1991.

[CT96] T. Chandra and S. Toueg. "Unreliable failure detectors for reliable distributed systems". Journal of the ACM, 43(2):225–267, Mar. 1996.

[DDS87] D. Dolev, C. Dwork and L. Stockmeyer, "On the minimal synchrony needed for distributed consensus", *Journal of the ACM*, 34(1): pp. 77-97, Jan 1987.

[DGG05] A. Doudou, B. Garbinato, and R. Guerraoui, "Tolerating Arbitrary Failures with State Machine Replication", In *Dependable Computing Systems*, John Wiley & Sons, Inc, 2005.

[DSU04] X. Defago, A. Schiper and P. Urban, "*Total order* broadcast and multicast algorithms: Taxonomy and survey", *ACM Computing Survey*, 36(4); 372–421, 2004.

[EMR01] P D Ezhilchelvan, A Mostefaoui, and M Raynal, "Randomized Multivalued Consensus", *In Proceedings of the Fourth International IEEE Symposium on Object oriented Real-time Computing (ISORC),* pp. 195-201, Magdeburg, Germany, May 2001.

[EZH08] Ezhilchelvan Paul, "Responsive Fault-Tolerant Computing in the era of Terascale Integration – State of Art Report" School of Computing Science, Newcastle University, Newcastle upon Tyne, UK. 2008.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process", *Journal of the ACM*, 32(2): 374-382, April 1985.

[FM97] Pesech Feldman and Silvio Micali. "An optimal probabilistic protocol for synchronous Byzantine agreement". *SIAM Journal on Computing*, 26(4):873-933, August 1997.

[GS97] R. Guerraoui and A. Schiper. “Software-based replication for fault tolerance”, *IEEE Computer*, 30(4), pp. 68–74, April 1997.

[IE06] Q. Inayat and P.D. Ezhilchelvan, “A Performance Study on the Signal-On-Fail Approach to Imposing *Total Order* in the Streets of Byzantium”, *In Proceedings of the 2006 International Conference on Dependable Systems and Networks*, pp. 578-587, Philadelphia, PA, USA, 25-28 June 2006.

[JS08] Habibullah Jamal, Kiran Sultan:, “Performance Analysis of TCP Congestion Control Algorithms”. INTERNATIONAL JOURNAL OF COMPUTERS AND COMMUNICATIONS Issue 1, Volume 2, 2008

[KS01] K. Kursawe and V. Shoup, “Optimistic asynchronous atomic broadcast." *Cryptology ePrint Archive*, Report 2001/022, Mar. 2001

[KV93] H. Kopetz, and P. Verissimo, “Real Time and Dependability Concepts.”In Mullender, S. (ed.), Distributed Systems, pp. 411-446. Wokingham : Addison-Wesley, 2nd ed., 1993.

[LAM01] L. Lamport “Paxos Made Simple”, *ACM SIGACT News (Distributed Computing Column)*, 32, 4: 18-25, December 2001.

[LAM06] L. Lamport, “Fast Paxos”, *Distributed Computing*, 19(2): 79-103. October 2006.

[LAM98] Leslie Lamport, “The part-time parliament”, *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[LSP82] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems*, 4(3): 382-401, July 1982.

[MF09] Mohammed. F. Ababneh, “Average Delay in TCP Networks”. American Journal of Scientific Research ISSN 1450-223X Issue 2 (2009), pp.5-9. Information Department, Prince Abdullah Ben Ghazi Faculty of Science and ITAl-Balqa' Applied University Assalt-Jordan.

[MJM08] Y. Mao, F. Junqueira, and K. Marzullo. “Mencius: Building efficient replicated state machines for WANs”. Technical Report CS2008-0930, Dept. of Computer Science and Engineering, UC San Diego, 2008.

[MNC+06] H. Moniz, N. F. Neves, M.l Correia and P. Veríssimo, “Randomized Intrusion-Tolerant Asynchronous Services”, *Proceedings of the International Conference on Dependable Systems and Networks* (DSN), Philadelphia, USA, pages 568-577, June 2006.

[NAG84] J. Nagle. RFC 896: “Congestion control in IP/TCP internetworks”. Jan. 1984.

[PSL80] M. Pease, R. Shostak, and L. Lamport, “Reaching Agreement in the Presence of Faults.” J. ACM, vol. 27 no. 2, pp.228-234, Apr. 1980.

[RIZ97] L. Rizzo. “Dummynet: a simple approach to the evaluation of network protocols”. *SIGCOMM Comput. Commun. Rev.*, 27(1):31– 41, 1997.

[SCH84] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors", *ACM Transactions on Computer Systems*, Vol. 2(2), pp. 145-154, May 1984.

[SCH90] F. Schneider. “Implementing fault-tolerant services using the state machine approach:” A tutorial. ACM Computing Surveys, pages 299–319, Dec. 1990.

[SCH93] F. B. Schneider. “Replication management using the state-machine approach”. In *Distributed Systems* (edited by S. Mullender), ACM Press, pp. 169–197, 1993

[UHS+04] P. Urbán, N. Hayashibara, A. Schiper and T. Katayama “Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm”, *In Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS’04)*, pp. 4-17, Ishikawa, Japan, October 2004.