2009

Performance Engineering of a Lightweight Fault Tolerance Framework

Hua Chai
Cleveland State University

PERFORMANCE ENGINEERING OF A LIGHTWEIGHT FAULT TOLERANCE FRAMEWORK

HUA CHAI

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

Taiyuan University of Technology

July, 2005

MASTER OF SCIENCE IN COMPUTER SCIENCE

Taiyuan University of Technology

July, 2007

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

December, 2009

This thesis has been approved for the

Department of **ELECTRICAL AND COMPUTER ENGINEERING**

and the College of Graduate Studies by

_____

Thesis Committee Chairperson, Dr. Wenbing Zhao

_____

Department/Date

_____

Dr. Yongjian Fu

_____

Department/Date

_____

Dr. Lili Dong

_____

Department/Date

To my loved mother and father

ACKNOWLEDGMENTS

I would like to thank the following people:

Dr. **Wenbing Zhao** for all his full-of-insight guidance as my supervisor, and for the virtues I learned from him.

Dr. **Yongjian Fu** and Dr. **Lili Dong** for their patience to convey the fundamental knowledge in the first semester.

Dr. **Dan Simon**, Dr. **Nigamanth Sridhar**, and Dr. **Changsu Yu** for their elaborately prepared lectures which imparted knowledge to me, and their perspectives of different things that enriched my view.

Robert Fiske for my improvement in English writing and better understanding of American culture.

Honglei Zhang, **Song Cui**, **Gang Tian**, and **Bo Chen** for their kind help and friendship.

I would also thank my mother. She supported me all the time.

PERFORMANCE ENGINEERING OF A LIGHTWEIGHT FAULT TOLERANCE FRAMEWORK

HUA CHAI

ABSTRACT

It is well-known that the Paxos algorithm can be used to build provably correct practical fault tolerant systems. In this thesis, a lightweight consensus framework - Paxos-Based Fault Tolerance (PFT) framework and its practical implementation is presented. It also includes how the system tolerates faults under practical conditions where the replicas might not be strictly homogeneous due to the asynchrony of their deployment environment. A comprehensive performance evaluation study is performed on the PFT framework. The approaches that can optimize the fault tolerance mechanisms under various practical scenarios are also discussed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYM

PFT Paxos-Based Fault Tolerance

BFT Byzantine Fault Tolerance

PBFT Practical Byzantine Fault Tolerance

MBQ Multicasting-Based Query

PBQ Primary-Based Query

Q/U Query/Update

HQ Hybrid Quorum

CHAPTER I

INTRODUCTION

In distributed multi-server systems, state machine replication is often used to ensure consistent state changes and outputs in response to a set of inputs. However, due to the existence of faults and failures, consensus is difficult and not always to be reached. In practical environment, messages can be delivered out of order, delayed for a long time, or lost during transmission; replicas may pause for a long period, fail to execute, and possibly restart. In a fault tolerant system, it is essential to prevent a fault from causing consensus failures, otherwise, it can lead to the divergence of state and confusing outputs to the clients and other components.

Consensus algorithms play an important role in state machine replication. Until now, a number of consensus algorithms and related fault tolerant frameworks have been proposed. The ones having been widely studied are the family of Byzantine Fault Tolerance (BFT) algorithms [1] and Paxos [2]. Due to the need to tolerate Byzantine faults, BFT is very costly. Once a client request is received by the leader, BFT protocols often involve two heavy communication steps. Paxos, however, incurs much less overhead, although it tolerates only benign faults. Paxos is very efficient in

common case(normal execution) where all replicas agree with a unique leader and the leader is not faulty. Due to its efficiency, Paxos has been used at Google's distributed Chubby locking service system [3].

Although the Paxos algorithm is well-described [4], it still requires system designers to make certain protocol extensions to apply it to practical systems [5]. In real world systems, the replicated servers might not be strictly homogeneous due to the asynchrony of their deployment environment. In particular, practical issues such as liveness and recovery must be addressed. For a practical fault tolerant system, how to operate efficiently under various non-desirable conditions deserves a great deal of attention. However, most research efforts on fault tolerant frameworks or consensus algorithms focus on optimizing the runtime performance under well-behaved system conditions. They mainly focus on analyzing and optimizing fault-free performance [1, 6] with few implementation and evaluation details reported [7]. Even though optimizing runtime performance for fault-free conditions is important, we believe that it is more crucial to design a fault tolerant system that operates gracefully under various faults and undesirable conditions. Such system is designed to cope with faults anyway.

In this thesis research, we focus on a number of important issues:

- How to select a new leader if the current leader fails.
- How to synchronize the state of a recovering replica.
- How to adjust the system parameters for optimal performance under various conditions.

The decision to focus on these issues is out of the following considerations:

(1) A robust leader election algorithm is necessary for the system to make progress under faulty conditions. According to Classic Paxos, the existence of a unique

leader is required to ensure liveness. If the leader crashes completely or cannot be accessible, a new leader needs to be chosen.

(2) A complete and efficient recovery mechanism is needed to keep state consistency among replicas. A replica missing order requests or client messages may result in the divergence of state and inconsistent outputs to clients. To prevent this risk, the system must help such a replica catch up and get state updated with no apparent effect on the system performance.

(3) Different systems may need different configurations. A fault-tolerance system should be able to get expected performance through selecting suitable parameters.

The research is carried on by implementing a lightweight fault tolerance framework —Paxos-Based Fault Tolerance (PFT) framework that is designed according to Classic Paxos [8]. The replicas can communicate with each other asynchronously and maintain consensus by tolerating benign faults under various practical conditions.

A significant advantage of the PFT framework is lightweight. It is simple to design because we choose not to tolerate Byzantine faults. It is unclear to us if we can achieve effective fault isolation in the presence of malicious adversaries that are determined to compromise the system. The other three important features of the replication protocol are that it is agreement-based, it is efficient for wide area networks [9] and it supports batching.

This thesis is organized as follows.

Chapter II provides background information and introduces related work. In particular, the suite of Classic Paxos algorithms are introduced. Chapter III describes the PFT framework. In chapter IV, we identify a number of practical issues facing our PFT framework. Two catch-up mechanisms and a novel view-change algorithm are discussed. The proof of correctness for the view change algorithm is also presented.

In chapter V, we report experimental results of the PFT framework. Chapter VI concludes the work for this thesis and provides future work.

CHAPTER II

BACKGROUND

2.1 Terminology

2.1.1 Consensus

To design a robust distributed fault-tolerant system, a fundamental problem to be solved is for all non-faulty replicas to reach consensus. Consensus can be seen as a process that requires a group of participants to agree on at most one single result. For example, replicas in a distributed database system need to agree on an operation for a given transaction; replicas in a file system need to maintain consistent file images; replicas in a flight control system need to maintain consistent state for any flight.

This problem can be difficult to deal with when one or more participants are faulty or their communication may experience failures [10]. If the systems mentioned above encounter consensus failures, the consequence may be extremely serious, therefore, a number of consensus algorithms have been proposed to ensure agreement among the participants despite failures [11].

2.1.2 Consensus Algorithm

The consensus issue has been of perennial interest among researchers in the field of distributed systems. Aiming to find the most optimal, provably correct solution to the consensus problem, numerous algorithms have been proposed [11]. They are designed for participants in a system to reach consensus with the presence of faults and communication failures.

A consensus algorithm needs to ensure two important properties: safety and liveness. The safety property ensures that consensus can be reached for each algorithm instance. The liveness property ensures progress of the system in the existence of faults.

The well-known consensus algorithms are BFT algorithms [1, 12, 13] and Paxos algorithms [2, 6, 14, 15, 16].

2.1.3 State Machine Replication

State machine replication [17] is a common approach to building fault tolerance frameworks. The core component of a consensus framework is the consensus protocol. A consensus protocol can be seen as the extension or application of a consensus algorithm for specific environment.

State Machine Replication Approach

"State machine replication is a well-known fault tolerance technique for building distributed services"[7, 17] and it is also a technique for converting a consensus algorithm into a fault-tolerant, distributed implementation. It is a general method for "implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. The approach also provides a framework for understanding and designing replication management protocols. Many protocols that

involve replication of data or software –be it for masking failures or simply to facilitate cooperation without centralized control –can be derived using the state machine approach[18]."

State machine replication can be designed according to the following rules [18, 19]:

1. Implement the same state machine on multiple, independent replicas.
2. Receive client requests and interpret them as inputs.
3. Choose an ordering sequence for the inputs and submit each input in an equivalent order at each state machine.
4. Execute each input in the chosen order on each state machine.
5. Respond to the client with the output.
6. Monitor differences in state or output of each state machine.

State machine replication requires the application to be deterministic. If two replicas start at the same state and receive identical sequences of inputs, they should end at the same state and produce identical outputs. Consensus is fundamental to state machine replication [17], so it is crucial to choose a robust consensus protocol.

Consensus Protocol

Consensus protocols are designed on the basis of consensus algorithms. Basically consensus protocols can be classified into two classes: quorum-based protocols and agreement-based protocols.

Quorum-based protocol

In the system with using such a protocol, only a quorum of replicated servers participate in ordering messages. Replicas are not required to communicate with each

other directly. When one or more replicas in the quorum crash or become inaccessible, it is necessary to add extra replicas from the rest of the group to reconstruct the quorum. Examples of quorum-based protocols are Query/Update (Q/U) [12] and Hybrid Quorum (HQ) [20].

Agreement-based protocol

The agreement-based protocol requires all non-faulty replicas to participate in ordering messages. It is based on multicasting and replica-to-replica communication. Such a protocol may become much less scalable than the quorum-based protocol as the number of faults tolerated by the system (thus the number of replicas) increases [12]. Another difference between this type of protocol and a quorum-based protocol is that the quorum-based protocol has the risk that the system can never expect when a quorum might fail, which may need extra communication steps to guarantee consensus; an agreement-based protocol forms its quorum dynamically during the ordering process and the system does not need to expect a quorum's failure as long as there is enough non-faulty replicas in the system.

2.2 Paxos

The Paxos algorithm is proposed in 1978, described and named in 1990, and published in 1998 [2]. This is an algorithm for solving consensus problem in tolerating benign faults in distributed environment.

The consensus process described in Paxos is for a group of participants to choose a single value. Paxos defines three roles for participants: proposers, acceptors and learners. Proposers propose a value, acceptors choose the value, and learners learn the value after the value has been chosen.

The assumption for the system to implement Paxos is that replicas can com-

municate with each other in asynchronous way. The system only needs to tolerant non-Byzantine faults: replicas may work at arbitrary speed and they may fail and restart; messages can be delivered arbitrarily long, can be lost, can be duplicated but they are never corrupted [8].

Paxos needs to guarantee the following requirements:

Safety

- Only a proposed value can be chosen.
- For each algorithm instance, only a single value can be chosen.
- Only a chosen value can be learned by a learner.

The Paxos algorithm executes in two communication steps:

Phase 1 (Prepare)

a. A proposer selects a sequence number n for its proposal and requests all acceptors to accept its proposal.

b. If an acceptor gets the proposal with a sequence number n higher than that of any proposal it has accepted, it accepts this proposal and promises that it will not accept any future proposal with a sequence number less than n. If an acceptor accepts this proposal, it responses to the proposer with the latest chosen value of the highest-numbered proposal.

Phase 2 (Accept)

a. If the proposer is informed that majority of acceptors have accepted the proposal with sequence number n, it picks a value for acceptors to choose. If the responses

from acceptors contain any chosen value, the proposer picks the value from the proposal with the highest sequence number; otherwise, the proposer can pick any value for acceptors to choose.

b. Upon receiving the proposed value for the accepted proposal with sequence number n, an acceptor needs to choose this value. Once the value gets chosen, it is informed to all learners. Because learners are not reliable, it may need any learner that learns the chosen value broadcasts the value to inform all other learners. A learner that misses to learn a chosen value needs to ask acceptors.

Paxos Family

To best apply Paxos in distributed environment, a family of algorithms based on the Classic Paxos has been developed, such as Fast Paxos [14], Cheap Paxos [15] and Fast Byzantine Paxos [6] Multicoordinated-Paxos [21] and Generalized Paxos [16].

Fast Paxos [14] is an optimized version of Classic Paxos in common case, which only requires two communication steps for participants to reach consensus. Although the optimized Paxos has fewer communication steps than Classic Paxos, it is not always the best choice for all deployment cases. Fast Paxos may suffer from collision problem which is not a concern for Classic Paxos. The performance will degrade if collisions occur. Furthermore, Fast Paxos might not perform better than Classic Paxos in wide area networks [9]. In wide area network, Classic Paxos has a significant probability of having a lower latency than Fast Paxos. Besides, to tolerate identical number of faults, Fast Paxos requires more replicas than Classic Paxos.

Disk Paxos is designed for implementing any fault-tolerant system composed of connected processors and disks. Like Classic Paxos and Fast Paxos, it tolerates non-Byzantine faults, but includes more detail on disk accessing. It claims that as long as the system exists only one non-faulty processor that can read and write majority

disks, Disk Paxos can guarantee progress for the system [22].

Cheap Paxos tolerates f faults with $f + 1$ main processors executing the system and f auxiliary processors only handling the failure of a main processor[15]. The main processors perform Classic Paxos while the auxiliary processors only work when a main processor fails. Cheap Paxos is implemented dynamically. When a main processor fails, the auxiliary processors reconfigure the system, remove the failed processor and reset quorums size in order to allow the remaining main processors proceed. Because of dynamical reconfiguration, Cheap Paxos is also referred to as Dynamic Paxos. After finishing the reconfiguration, the main processors resume execution.

Fast Byzantine Paxos [6], also referred to as Byzantine Paxos protocol, is designed for improving the common case performance while tolerating Byzantine faults. It requires fewer steps than a general BFT protocol, therefore, it can be seen as an improved BFT protocol.

2.3 Related Work

The Paxos algorithm [2] is the foundation of this thesis work. We adapted the algorithm for state machine replication and made a number of optimizations not seen in the original algorithm.

Google first uses Paxos at its distributed Chubby locking service system [3]. Chubby is a fault-tolerant system and has been used by several Google systems, such as the Google File system and Bigtable which are also called Chubby clients. Like Chubby, our framework also uses replication to tolerate faults and chooses a leader among those replicas. Chubby helps clients to find the leader and it is the leader to serve all client requests. In our framework, all replicas including the leader serve all client requests and clients do not need find the leader. Requests from clients can

be guaranteed to reach the leader by multicasting-based communication. In other word, the replication system is almost transparent to clients. If the leader fails, a new leader is automatically elected. Chubby requires replicas to compete for a lock to become the leader, while in our framework we use dynamic view change to solve leader election problem. Chubby uses coarse-grained locks and claims that they impose much less load on the lock server than fine-grained locks do. In this aspect, our framework is also less load structure because it only replaces a leader as it fails. As Chubby elects a new leader, it takes long Chubby outages, the availability of which is hard to be improved due to its using coarse-grained locks. In our work, we try to reduce heavy recoveries and use lightweight catching up instead.

Tushar Chandra [5] and his partners describe their experience in building a fault-tolerant database system into Chubby using the Paxos consensus algorithm. Chubby uses this database to store its state. Except the work mentioned in [3], they implement a extra fault-tolerant log for their system in addition to the database log which is distributed among replicas. The fault-tolerant log records all consensus protocol actions locally and persistently. Replicas can use this log to reconstruct state information when they re-join and also use this log for lagging replicas to catch up their state. They claim that it only needs a single disk write to each replica's log per protocol instance. Their fault-tolerant log is similar to the state log described in this thesis, which is replicated over all replication servers. The difference is that the state log records only state history for ordered messages(including ordering number, ordering state and message information) not algorithm actions. To prevent critical log writes from unexpected delays, in the implementation part we choose asynchronous disk write to update our state log. Like the replication log in [5], replicas truncate their state logs as a snapshot is informed but in our framework the snapshot is taken by replicas not clients and there is no snapshot synchronization problem. In this

paper [5], they also demonstrate some engineering problems encountered including disk corruption, leader status loss. For disk corruption, they use a marker to identify if a replica has a corrupted disk. We use different recovery strategies from that used in the article. In our framework, we combine view change and checkpoint algorithms to help a rebuilding replica to determine its state. The only common aspect for this part is that it allows a recovering replica to participate in Paxos as a non-voting member. In our framework we deal with rebuilding or lagging replicas in the similar way. In [5], if a leader lost its status, it cannot regain it again. This can be guaranteed by using a global epoch number. In our work, we use global view identification and view change algorithm for our framework to solve this problem. A leader cannot request other replicas to accept it in two different views. An amphibious leader will be finally replaced by view change. The work in [5] contains runtime consistency checking. However, it is more based on testing purpose. In our replication system, we provide runtime consistency checking which not only guarantees the safety of the system but works with catching up strategies. Our work also includes reducing leader failovers and runtime overhead caused by software bugs which are also considered in the Tushar's work.

Other related work to this thesis is about catch-up mechanism and view-change algorithm. They can be summarized as follows:

PBFT framework [1] uses a view change algorithm for leader choosing as well. Different from PBFT, the view change algorithm used in PFT framework is optimized. It costs less for installing a new view. As for dealing with recovery problems, PBFT [1, 23] mainly focuses on message retransmission and state transfer. It requires replicas periodically share message information to get missing client messages back. Through this approach, if a replica notices that other replicas may have lost messages, it retransmits the missing messages to them. If a replica cannot get missing messages

retransmitted from other replicas, PBFT requires the replica to ask a retransmission from the client. A replica that cannot get messages back by both ways needs a state transfer to catch up. This work mostly focuses on client message loss and retransmission instead of ordering messages.

Another relevant work is in Zyzzyva/Zyzzyva5 [13] that allows the replica finding a hole in its $ORDER-REQ$ message history to send a $FILLHOLE$ message to the primary. The $FILLHOLE$ message is used to request the retransmission of missing $ORDER-REQ$ messages. After sending out a $FILLHOLE$ message, if the replica cannot receive the missing messages within a given period of time, it broadcasts the $FILLHOLE$ message to all replicas for retransmission of the missing messages. However, it does not present much evaluation on the algorithm's performance.

The Q/U protocol [12] uses its replica history to do recovery. The replica history is used by replicas to retain operation-related information. Replicas are allowed to share their latest replica histories with clients. If clients find that replicas reach to different state by detecting replica histories, they need to bring the replicas into a consistent state. However, this recovery approach is too tied to the client. In our work, we don't want the replication recovery to rely on any client and try to reduce the dependency between the replica and the client.

CHAPTER III

PFT FRAMEWORK

3.1 Framework Design

In the PFT framework, an adapted version of Paxos protocol is used to implement state machine replication. A state log is designed to record state information in each state machine.

3.1.1 System Model

We assume that the applications running on the top of the PFT framework follow the client-server interaction model where a client issues a request to the server replicas and waits for the corresponding reply before it issues a new one. All processes (including clients and server replicas) may be subject to non-malicious faults, and the network may lose messages. We assume that there are $2f + 1$ replicas available to tolerate up to f faulty replicas. There is no restriction on the number of faulty clients. One of the replicas is designated as the primary and the remaining ones as the backups. Furthermore, we assume that the replicas execute deterministically. How to

properly handle replica nondeterminism is important, however, it is out of the scope of this thesis research.

3.1.2 Replication Protocol

In the PFT framework [24], an adapted version of Paxos protocol is used. The replication protocol executes in two-phases during normal operation. In this protocol, the primary (the leader of the replicas) plays the role of the unique proposer. Why the protocol allows the unique proposer instead of multiple proposers will be explained in the next chapter. In a nutshell, the protocol works as follows. When the primary gets a request message from a client, it assigns the request a total order with a sequence number and prepares an ordering request for this message. Once a replica (including the primary itself) gets an ordering request, it accepts the designated order in the ordering request (acting as an acceptor). When the primary learns that the majority of replicas ($f + 1$) have accepted the designated order, the primary notifies all other replicas, which would act as learners, to commit to the designated order by sending another ordering request. Once an order is committed, the corresponding request message is ready to be executed by the application.

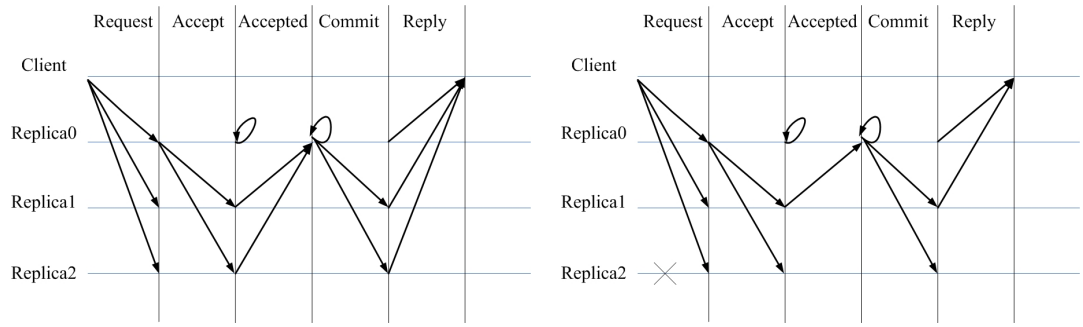
The detailed operations of the replication protocol are described below (assume the primary will not crash).

Accept phase

a. When it receives a non-duplicate request from a client in the form

   $< REQUEST, c, m, data >$, the primary sends an $ACCEPT$ request

   $< ACCEPT, n, c, m >$ to all replicas, where c is the client identification number, m is the client message number, and n is the sequence number representing the total order of the request.



(a) Replication algorithm under normal operation.

(b) Replication algorithm with one faulty replica.

Figure 1: Protocol execution.

b. Upon receiving an $ACCEPT$ request $< ACCEPT, n, c, m >$, a replica accepts the order with the sequence number n if it has not accepted an order with a higher sequence number than n and promises that it will never accept any order with a lower sequence number than n. If the replica accepts this order, it responds to the primary that it has accepted this order by sending a $ACCEPTACK$ message $< ACCEPTACK, n, i >$, where i is the replica identification number.

Commit phase

After sending out a $ACCEPT$ request $< ACCEPT, n, c, m >$, the primary keeps collecting $ACCEPTACK$ responses to the $ACCEPT$ request. If the primary collects the $ACCEPTACK$ responses $< ACCEPTACK, n, i >$(including the one the primary sent out) from the majority of replicas, it sends another ordering request - a $COMMIT$ request $< COMMIT, n >$ to all replicas. On receiving this $COMMIT$ request, a replica needs to commit to this order. Once the order is committed, the corresponding request message is ready to be executed by the application.

Figure 1(a) and Figure 1(b) show the operations of the replication protocol with and without faulty replica existing, separately. In the figures, there are three replicas

($2f + 1 = 3$) available to tolerate up to one ($f = 1$) faulty replica. It requires that there must be two replicas ($f + 1 = 2$) to form the quorum(a group composed of the majority of replicas). At the accept phase of both figures, when the primary ($Replica_0$) receives a client request, it assigns the request a total order with a sequence number and prepares an $ACCEPT$ request for this message, then it broadcasts the $ACCEPT$ request to all replicas. Once a backup ($Replica_1$ or $Replica_2$) accepts this order, it sends an $ACCEPTACK$ response to the primary. If there is no faulty replica (Figure 1(a)), the primary can form the quorum with any backup ($Replica_1$ and $Replica_2$). If there is a backup ($Replica_2$ in Figure 1(b)) that crashed before the quorum is formed, the primary still can form the quorum with $Replica_1$ as long as the primary can get the $ACCEPTACK$ response from $Replica_1$. Once the primary learns that the majority replicas (the primary and $Replica_1$) have accepted this order, it sends a $COMMIT$ request to notify all replicas to commit to this order. According to the two figures, the same case is for the replication protocol to tolerate up to any f faulty replicas. However, it must be based on the premise that the primary is alive during the whole ordering process.

3.1.3 Data structures

Request queue: the data structure is used to store client requests. Once a replica receives a client request, it inserts the request into the request queue. Certificate: the data structure is used for the primary to record accept decision from replicas for a total order. When the primary prepares a total order for a client message, it creates a certificate for this order. When the primary gets an $ACCEPTACK$ response, it records the response in the certificate. The primary can learn if an order can be committed or not by checking the corresponding certificate. Order table: the data structure is used to store a set of order records. Each order record contains the basic

information for a total order: the sequence number and the corresponding request message for the total order. Once a replica accepts an order, it constructs an order record and inserts it into the order table. Response queue: the data structure is used to store responses to clients. Once a replica executed a client request, it generates a response to the request and inserts the response into the response queue. If later on the replica receives this request again, it gets the corresponding response from the response queue and resends it to the client.

3.1.4 State Log

The state log, built in each replica, is an important component of the PFT framework. The log records ordering information and state information for orders. The state for each order has two values: A and C. A represents that the corresponding order has been accepted and C represents that the order has been committed.

In an ordering process, when a replica steps into another state, it first needs to log the state for the order. According to our replication protocol, a complete ordering process requires each replica to have at most two writes to its state log. Once a replica accepts an order, it records the order's information in its state log and denotes the state of the order as A. Later on if the replica commits the order, it gets the order in its state log and updates the state of the order as C.

An order's state in a replica's state log is recorded as C if and only if this replica has committed this order. If an order in the state log is not recorded as C, the replica may still participate in this ordering or give up this order before commitment. It is very useful for fast tracing an ordering process and checking if there is any uncommitted or missing order in a replica. As an important tool in fault tolerance, a replica can use its state log for a recovery after restart from failure or use other replicas' state logs for a catch up after missing part ordering information. This will

be discussed in the next chapter in detail.

In the PFT framework, we use the state log to only serve the protocol. The state log only records state information for orders instead of any information of application. This is because we want to better separate the boundary of the protocol and the application and allow the protocol to achieve simplification and efficiency. If the application needs to maintain application related state, it can have a separate application log. The separation of the protocol and the application level will be efficient for the fault-tolerant framework design.

3.2 Features

A significant advantage of the PFT framework is being lightweight. It is simple to design and does not need to tolerate complex faults, such as the BFT faults. The other three important features of the replication protocol are that it is agreement-based, it is efficient for wide area networks and it supports batching.

(1) In the PFT framework, the protocol is agreement-based. In such a protocol, the communication is replica-to-replica broadcast and all non-faulty replicas are required to be involved in all ordering process. Each replica decreases useful work as the faults expected to tolerated increases [12]. However, compared with other agreement-based protocols such as Practical Byzantine Fault Tolerance (PBFT), our replication protocol can cause less overhead. The reason is that for common case only the channels between the primary and backups are active while the channels between backups are idle.

(2) The PFT framework is designed based on Classic Paxos that is more efficient than the protocols with *client* $\rightarrow$ *replica* $\rightarrow$ *client* communication patten for wide area networks [9]. The ordering latency of the PFT framework is determined

by the transmission time from the client to the primary while for the protocols with *client* $\rightarrow$ *replica* $\rightarrow$ *client* communication patten, such as Fast Paxos, the ordering latency is greatly determined by the transmission time from the client to the slowest replica in the protocol's quorum.

(3) The PFT framework allows the primary to collect a group of messages for an order. With batching, the system can dramatically improves its throughput.

CHAPTER IV

PFT UNDER CONTROL

In this chapter, we identify a number of challenging scenarios in which the PFT framework could be used. We analyze the technical issues involved with the scenarios and provide our solutions.

4.1 Batching

4.1.1 Problem

Although the two-phase message ordering could be executed concurrently, starting a new round message ordering process for each client request is still inefficient. As one can envisage, if the framework could order a batch of requests at a time, the throughput could be significantly increased.

4.1.2 Solution

To support matching, the primary uses a general buffer (referred to as the *batch*) to log the requests to be ordered. When batching is enabled, the primary will order

the *batch* each time instead of a single request. The requests stored in a *batch* must be independent of each other, which means that the messages in the same batch cannot come from the same client. This policy helps achieve fairness.

Ideally, the primary should order a full batch of requests at a time. However, insisting on collecting a full batch before starting the ordering process is not realistic because the message transmission time is variant and some messages may arrive very late at the primary. To avoid this problem, we introduce a batch timer. If the primary cannot collect enough messages to fill the *batch* before the timer expires, it initiates an ordering request for the requests collected in the *batch* so far and resets the timer.

4.1.3 Algorithm Design

Our replication protocol is modified in the following way to enable batching.

Accept phase

a. When it receives a non-duplicate request from a client in the form

   $< REQUEST, c, m, data >$, the primary sends an $ACCEPT$ request

   $< ACCEPT, n, S >$ to all replicas, where c is the client identification number, m is the client message number, n is the sequence number representing the total order of the request, and S is a set of independent messages stored in the *batch*.

b. Upon receiving an $ACCEPT$ request $< ACCEPT, n, S >$, a replica accepts the order with the sequence number n if it has not accepted an order with a higher sequence number than n and promises that it will never accept any order with a lower sequence number than n. If the replica accepts this order, it responds to the primary that it has accepted this order by sending an $ACCEPTACK$ message $< ACCEPTACK, n, i >$, where i is the replica's identification number.

Commit phase

After sending out an $ACCEPT$ request $< ACCEPT, n, S >$, the primary keeps collecting $ACCEPTACK$ responses $< ACCEPTACK, n, i >$(including the one the primary sent out) from different replicas until it learns that the majority of replicas have accepted this order. Once the primary learns that the majority of replicas have accepted this order, it broadcasts a $COMMIT$ request $< COMMIT, n >$ to all replicas. On receiving this $COMMIT$ request, a replica commits to this order. Once the order is committed, the corresponding request messages are ready to be executed by the application.

4.2 Garbage Collection and Checkpoint

4.2.1 Problem

Extensive execution of the PFT framework can cause enormous consumption of resources and thereby degrading the system performance significantly. To prevent the shortage of available resources and the heavy load on the system, it is necessary to introduce garbage collection mechanism into our framework. Through garbage collection, we can eliminate stored messages and other data structures. However, when and how a replica carries out the garbage collection for stored messages and data structures is non-trivial [24]. A replica cannot arbitrarily delete requests or data structures stored for committed orders because they might be needed for a slow replica to catch up or for a new replica to join the system.

4.2.2 Solution

The problem for garbage collection can be solved by the checkpoint mechanism for the PFT framework [24]. A checkpoint of a replica contains both the snapshot

of the application state and that of the middleware state (i.e., the state maintained for replication). The checkpoint can be taken after the execution of every application request, however, doing so would be prohibitively expensive. Therefore, our frame initiates a checkpoint periodically after a group of messages have been executed. When a replica has ordered and executed a certain amount of messages, it takes a checkpoint of both the application state and the middleware state.

The messages and data structures stored before this checkpoint can be garbage collected if the replica learns that the majority of replicas have all taken such a checkpoint. This checkpoint is referred to as a stable checkpoint. If a backup replica fails to receive some ordering messages and the messages have been garbage collected by other replicas, the replica can request the latest stable checkpoint from a correct replica to roll forward to the state reflected in the checkpoint. The process of a replica receives and installs a stable checkpoint from other replicas is referred to as a state transfer.

4.2.3 Algorithm Design

In the PFT framework, a checkpoint is periodically taken after executing a certain amount of messages. The number of the messages should be determined by the specific system. The algorithm for the checkpoint mechanism is described below.

When a replica gets a checkpoint, it broadcasts a $CHECKPOINT$ message $< CHECKPOINT, l, i >$ to notify all other replicas about its latest state. In this $CHECKPOINT$ message, l is the sequence number representing the total order of the request executed before the checkpoint is taken, also referred to as the checkpoint number, and i is the replica's identification number.

When a replica collects $CHECKPOINT$ messages from the majority of replicas

with the same checkpoint number l, the replica considers the checkpoint l as a stable checkpoint. When a replica learns about the stable checkpoint, it can discard the collected messages and data structures up to this checkpoint. A slow replica could also take this opportunity to catch up with other replicas by requesting a state transfer and installing the latest checkpoint.

4.3 Catch-Up Mechanism

4.3.1 Problem

Due to the possibility of overload, message delay and message lost, some replicas may lag behind the others. In this case, a lagging replica can catch up with other replicas by requesting a state transfer. After installing a stable checkpoint, a replica's state may get much closer to that of the primary. However, an overload situation may cause frequent state transfers. Because the state could be large, especially for complicated applications, catching up via state transfer is not always recommended. For some replicas, they may have temporarily missed a few client requests or ordering messages. In these cases, designing a more efficient method to obtain the missing information becomes important.

4.3.2 Solutions

Active Approach

Each replica periodically checks its state log. It shares ordering information with other replicas. Once a replica finds a reporting replica that has missed ordering requests, the receiver checks the sequence numbers of the missing requests and compares them with its stable checkpoint number l. For those orders with higher sequence number than l, the receiver constructs a set of state records and sends them to the

reporting replica. For those orders with lower sequence number than l, the replica, if it is the primary, sends its latest stable checkpoint to the reporting replica. On getting missing ordering information back, a replica updates its state log and keeps processing messages.

Passive Approach

Unlike the active approach, replicas find missing ordering information not by periodically checking state logs and exchanging ordering information with each other. A replica would realize that it has missed some ordering messages when it receives an ordering request with higher sequence number than that it is expecting. In this case, a hole is formed in the replica's state log due to the missing ordering messages. When the replica notices the existence of the hole, it will try to fill the hole by obtaining the missing ordering information from the primary. Once the primary gets a request from a reporting replica that has missed ordering messages, the primary checks the sequence numbers of the missing messages and compares them with its latest stable checkpoint number l. For those orders with higher sequence number than l, the primary constructs a set of state records and sends them to the reporting replica. If those orders carry lower sequence number than l, the primary sends its latest stable checkpoint to the reporting replica. On getting missing ordering information back, a replica updates its state log and resumes executing messages.

4.3.3 Algorithm design

In the PFT framework, we design two catch up algorithms according to the active and passive approaches, respectively. The one designed according to the active approach is called Multicasting-Based Query (MBQ) while the other is referred to as Primary-Based Query (PBQ).

MBQ

The MBQ algorithm is described below. Each replica periodically checks its state log. If a replica notices that it may have missed one or more ordering messages ($ACCEPT$ or $COMMIT$), it broadcasts a query order message to all replicas $< QUERYORDER_{Multicast}$, n_{lower}, n_{upper}, $i >$, where n_{lower} is the lower bound of the sequence number of the missing requests, n_{upper} is the upper bound of the sequence number of the missing requests, and i is the replica identification number.

Sometimes, a replica might not know it has lost the most recent one or more ordering messages. In this case, it is desirable to have a replica to still send a query order message to other replica reporting its status. In the query order message, n_{upper} is set to the value of the latest accepted order's sequence number, and n_{lower} is set to an invalid value.

When a replica receives a $QUERYORDER_{Multicast}$ message, it first checks the lower bound n_{lower}. If n_{lower} is an invalid value, the replica compares n_{upper} with its latest committed order's sequence number n_{gid}. If n_{upper} is less than n_{gid} by a certain value (according to the message transmission and processing time), which means that the reporting replica may have missed the ordering messages with sequence number between n_{upper} and n_{gid}, it sets n_{lower} to the value of n_{upper} and n_{upper} to the value of n_{gid}. The receiver then compares its stable checkpoint number l with n_{lower}. There are two situations:

(1) If n_{lower} is a valid value and the value is larger than l, the receiver can only construct state records corresponding to the orders with sequence numbers between n_{lower} and n_{upper}, and then sends a response order message

$< RESPONSEQRDER, i, S >$ to the reporting replica, where i is the sender's identification number, and S is a set of state records constructed according to the order records with the sequence number ranging from n_{lower} through n_{upper}.

Each state record consists of a tuple $< state, n, c, m >$, where n is the sequence number representing the total order of the request, and c and m uniquely identify the client request for the order with sequence number n (If batching is enabled, each state record should consist of a tuple $< state, n, MS >$, where MS is a set of messages in the batch for the order with sequence number n).

(2) If n_{lower} is a valid value but less than the receiver's latest stable checkpoint number l, the replica, if it is the primary, initiates a state transfer to the reporting replica, and then sets n_{lower} to $l + 1$. If n_{lower} is less than or equal to n_{upper}, the receiver repeats step 1.

Once a replica gets a $RESPONSEQRDER$ message with a set of state records, it processes the received records, updates its state log and resumes executing messages.

PBQ

The PBQ algorithm is described below. If a replica receives an ordering message ($ACCEPT$ or $COMMIT$) with higher sequence number than it expects, the replica sends a query order request $< QUERYORDER_{Primary}$, $n_{expected}$, $n_{received}$, $i >$ to the primary, where $n_{expected}$ is the sequence number of the next ordering message that the replica expects to receive, $n_{received}$ is the sequence number of the most recent ordering message that the replica received, and i is the replica's identification number. When the primary receives a $QUERYORDER_{Primary}$ message, it compares its latest stable checkpoint number l with $n_{expected}$ in the message. There are two situations:

(1) If $n_{expected}$ is larger than the primary's latest stable checkpoint number, the primary only constructs state records corresponding to the orders with sequence numbers between $n_{expected}$ and $n_{received}$, and then sends a response order message $< RESPONSEQRDER, i, S >$ to the requester, where i is the sender's identification number, and S is a set of state records constructed according to the

order records with the sequence number ranging from $n_{expected}$ through $n_{received}$. Each state record consists of a tuple $< state, n, c, m >$, where n is the ordering number, and c and m uniquely identify the client request for the order with sequence number n (If batching is enabled, each state record should consist of a tuple $< state, n, MS >$, where MS is a set of requests in the batch for the order with sequence number n).

(2) If $n_{expected}$ is less than the primary's latest stable checkpoint number l, the primary initiates a state transfer for the requester, and then sets $n_{expected}$ to $l+1$. If $n_{expected}$ is less than or equal to $n_{received}$, the primary repeats step 1.

Once a replica gets a $RESPONSEQRDER$ message with a set of state records, it processes the received records, updates its state log and resumes executing messages.

4.4 View Change Mechanism

4.4.1 Problem

Just like other components in a fault tolerant system, the primary may fail as well. To ensure continuous operation, it is essential to elect a new leader when the current primary fails. This process is referred to as a view change, and it is the focus of this section. The view change mechanism is instrumental to ensure the liveness property of our replication protocol.

4.4.2 Solutions

For the Paxos algorithm, it is desirable to have a unique proposer to issue proposals [8]. This can prevent multiple proposers from issuing competing proposals at the same time, which may result in the inability of choosing any value at all. However,

the unique leader constitutes a single point of failure. If the leader fails, it is necessary to dynamically elect a new leader to ensure further progress.

Before presenting the view change approach, we first discuss how to identify the leader. In designing the replication protocol introduced in the previous chapter, we have assumed that there is a default unique leader (i.e., the primary) known to all replicas in the system. If the primary fails, the rest of replicas must elect a new primary. We adopt the approach introduced in [1], which uses a view to represent a configuration with a unique leader. In this approach, the leader election is replaced by a view change. When the group of replicas accept and install a new view, a unique leader is naturally chosen. Each view is assigned a view number, which is monotonically increasing. The leader in a view v is determined to be the replica with an id $i = vmodN$, where N is the total number of replicas.

We extended the replication protocol by incorporating the support of view changes. The main change involves with the ordering messages is the addition of a parameter indicating the current view. For each view, the primary's identity is implicitly determined.

The view number is not only used for replicas to identify the primary but to protect the safety of the protocol in the event of primary failures as well. If some replicas agree on another replica as the primary, it may cause that different replicas decide on different orders for a message request. It would lead the system to state inconsistency. To prevent this situation from happening, the protocol requires replicas to exchange ordering messages with their view numbers included. A replica is forbidden to accept or commit to any order from a deferent view. In this case, a replica at a different view cannot disturb the ordering process of the current view because its messages will not be processed by the majority of replicas. A replica in an old or incorrect view must be notified and brought up to date, however.

The primary could crash at any time. Once the primary crashes, the other replicas must dynamically move to a new view (and hence elect a new primary). The view change mechanism uses a timeout to trigger a view change. In our framework, if a replica cannot commit to an order before a view change timeout, the replica will initiate a view change. Different systems should have different values for the view change timer, which cannot be too short or too long. If the timeout is set too short, any backup experiencing a message delay or message loss can trigger a view change even if the primary is still accessible. If the timeout is set too long, it might take too long for a replica to detect the failure of the current primary, which will reduce the system's availability.

If one of the replicas initiates a view change, a voting procedure is conducted among the replicas. The result of the voting is that a new primary eventually gets chosen, i.e., a new view is installed. Figure 2 shows the view change procedure($Replica_0$ is the primary and cannot reachable), which consists of two phases described below.

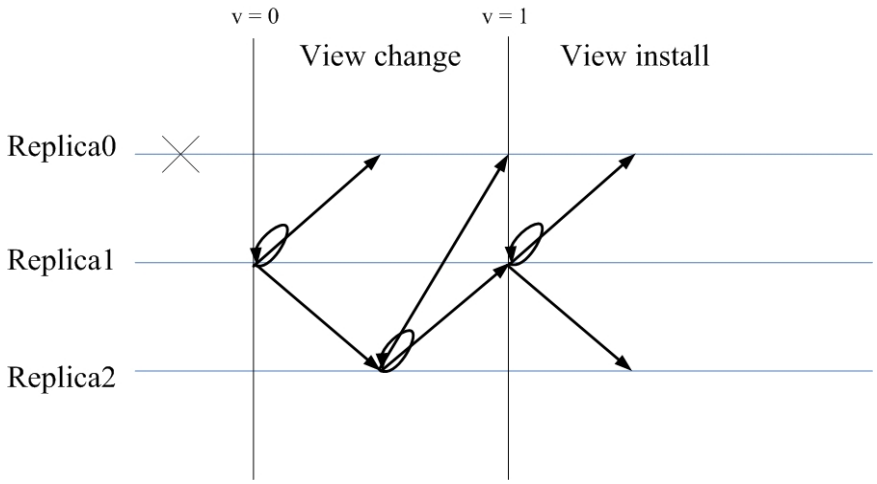


Figure 2: The view change algorithm for the PFT framework.

Phase1

a. A replica that cannot commit an order on expiration of the view change timer initiates a view change by broadcasting a view change request to all replicas.

b. Each replica receiving such a view change request also suspects the primary by broadcasting a view change request, if it has not done so already.

Phase2

The primary in the new view has collected view change requests (including the one it has sent out) from the majority of replicas, it installs the new view and sends a new view request to notify all replicas to install the new view.

The view change mechanism requires that a replica only can execute view change, new view requests and checkpoint messages during a view change process. The system can resume normal execution only after the view change process has finished.

If a new primary gets chosen, it has to find out what application requests have been ordered by the old primary and reissue ordering requests for them according to the same order.

4.4.3 Algorithm design

With the view number included, the protocol is extended as follows:

Accept phase

a. When it receives a non-duplicate request from a client in the form

$< REQUEST, c, m, data >$, the primary sends an $ACCEPT$ message in the form $< ACCEPT, v, n, c, m >$ (or $< ACCEPT, v, n, S >$ if batching is enabled, the same is true for all other messages and therefore this difference will be omitted from now on) to all replicas, where c is the client identification number, m is the client message number, v is the replica's view number, and n is the sequence number representing the total order of the request.

b. Upon receiving an $ACCEPT$ message, a replica accepts the order with the sequence number n if it has not accepted an order with a higher sequence number than n. If the replica accepts this order, it responds to the primary that it has accepted this order by sending a $ACCEPTACK$ message in the form $< ACCEPTACK, n, i >$, where i is the replica's identification number.

Commit phase

After sending out the $ACCEPT$ message for this order, the primary keeps collecting $ACCEPTACK$ responses from different replicas until it learns that the majority of replicas (i.e., $f + 1$) (including itself) have accepted this order in the same view v, then it sends a $COMMIT$ message in the form $< COMMIT, v, n >$ to other replicas so that they can learn the chosen order. On receiving the $COMMIT$ message, a replica in the same view v is required to commit this order with the sequence number n, after which the message for this order can be executed.

In the extended protocol, the view change algorithm for the PFT framework is described below.

View change

a. On expiration of the view-change timer, a backup replica broadcasts a

$VIEWCHANGE$ message $< VIEWCHANGE, v+1, n, l, P, i >$ to all replicas and set its view change flag, where i is the backup's identification number, $v+1$ is the new view number, n is the sequence number representing the total order of the last committed request, l is the last stable checkpoint number, P is a set of records on accepted or committed records. Each record consists of a tuple $< view, n, c, m, rt >$, where $view$ $(<= v)$ is the view number of the record, c

is the client identification number, m is the message number, n is the sequence number representing the total order of the request, and rt is a flag indicating if it is an accepted or committed record.

b. Once a backup receives a $VIEWCHANGE$ request and has not set its view change flag, it sets the flag and broadcasts a $VIEWCHANGE$ request to all other replicas. Otherwise, it ignores the message.

c. The primary in view $v+1$ records the $VIEWCHANGE$ messages received.

View installation

If the primary in view $v+1$ can collect $f+1$ view change messages from different replicas for the view $v+1$, it installs the new view and notifies all backups by broadcasting a $NEWVIEW$ message in the form $< NEWVIEW, v+1, O >$, where O is a set of messages determined in the following way:

For each sequence number n between the largest stable checkpoint number l_{max} and the largest reported committed sequence number n_{max}:
if the new primary has received a corresponding $COMMIT$ message (i.e., one with a sequence number n)
$then$ it constructs a new $COMMIT$ message $< COMMIT, v+1, n >$;
$elseif$ it has not received the corresponding $COMMIT$ message
$then$ it searches for the corresponding $COMMIT$ message from the received $f+1$ $VIEWCHANGE$ messages;
if such a $COMMIT$ message is found
$then$ it constructs a new $COMMIT$ message $< COMMIT, v+1, n >$ in the new view;
$elseif$ a corresponding $ACCEPT$ message is found, it constructs a new $ACCEPT$ message $< ACCEPT, v+1, n, c, m >$ in the new view.

else it constructs a null $ACCEPT$ message. The execution of the null application request is a no-op. Once a backup receives the $NEWVIEW$ request, it installs the new view and updates its ordering information.

During a view change, replicas includes their stable checkpoint number in their view change requests. This checkpoint number tells the new primary if a replica needs a state transfer. If a replica's stable checkpoint number in its view change request is lower than that stored for the new primary, the new primary will send its latest stable checkpoint to the replica.

4.4.4 Proof of Correctness

We prove the correctness of the view change algorithm by contradiction. Assume that replica i committed a message m with sequence number n in view v, but replica j committed the same message with a different sequence number n'!= n in view u. If $v = u$, the normal operation of our replication algorithm ensures n '= n, which contradicts the assumption. Without loss of generality, we assume $v < u$. The commit message for sequence number n in view v might not have reached all non-faulty replicas before the primary crashed. We consider two circumstances based on if the primary of view u has received the commit message for sequence number n and message m.

(1) The primary in view u received the commit message with sequence number n for m while it was in view v, or it found a corresponding commit record in the view change messages during the view change to u. According to our view change algorithm, the primary in view u will reissue a commit message for m with the same sequence number n. All non-faulty replicas would then execute m according to the sequence number n. If replica j in fact committed m with sequence number

n', it must have received a reissued commit message for m with sequence number n', which is impossible.

(2) The primary in view u never received the commit message with sequence number n for m, and it did not find a corresponding commit record during the view change to u. Because replica i committed m with sequence number n in view v, at least a set R1 of $f + 1$ replicas must have received the accept message with sequence number n and responded to the primary in view v. During the view change to u, the primary in u must have collected $f + 1$ view change messages from a set R2 replicas. Because there are total $2f + 1$ replicas, R1 and R2 must intersect in at least 1 replica. If replica j committed m in view u with a sequence number n', it implies that the primary reissued an accept request for m with a sequence number n'. This is impossible because for this to happen, (1) either the replica in the intersection did not pass the accept record for m with sequence number n to the primary in view u, or (2) the primary in view u did not reissue an accept message for m with the sequence number n. Neither would happen according to our view change algorithm.

4.4.5 Optimization

According to our view change algorithm, if a backup fails to commit to an order, it will trigger a view change and eventually replace the current primary for a new one no matter whether the primary is alive or not. However, some view changes may be caused by the loss of an ordering message issued by the primary (instead of a primary failure). Obviously, this kind of view changes are not necessary and should be avoided. Out of this concern, the view change timer is temporarily disabled when a replica is engaging a state transfer or is in the process of catching up with other replicas.

Because the primary largely controls the speed of message ordering, and hence the system performance, it is desirable to proactively induce a view change when the current primary exhibits poor performance before the expiration of the view change timer at the backup. To implement this mechanism, the system throughput is measured continuously by the replicas, if the throughput drops below a predefined value, a backup will initiate a view change so that another replica could serve as the primary in the new view.

4.4.6 State Transfer

The primary replica may receive a state transfer request at any point in time between two successive checkpoints. If the request arrives soon after it has taken a checkpoint, the primary transfers this latest checkpoint to the lagging replica. The lagging replica may have a good chance to catch up with other replicas by applying the checkpoint. However, if the state transfer request comes too late, the lagging replica might not be able to fully catch up until the next checkpoint time (because it has to process all queued application requests since the last checkpoint). It will be more practical to delay responding to the state transfer request after the primary has taken the next checkpoint. The two scenarios are illustrated in Figure 3.

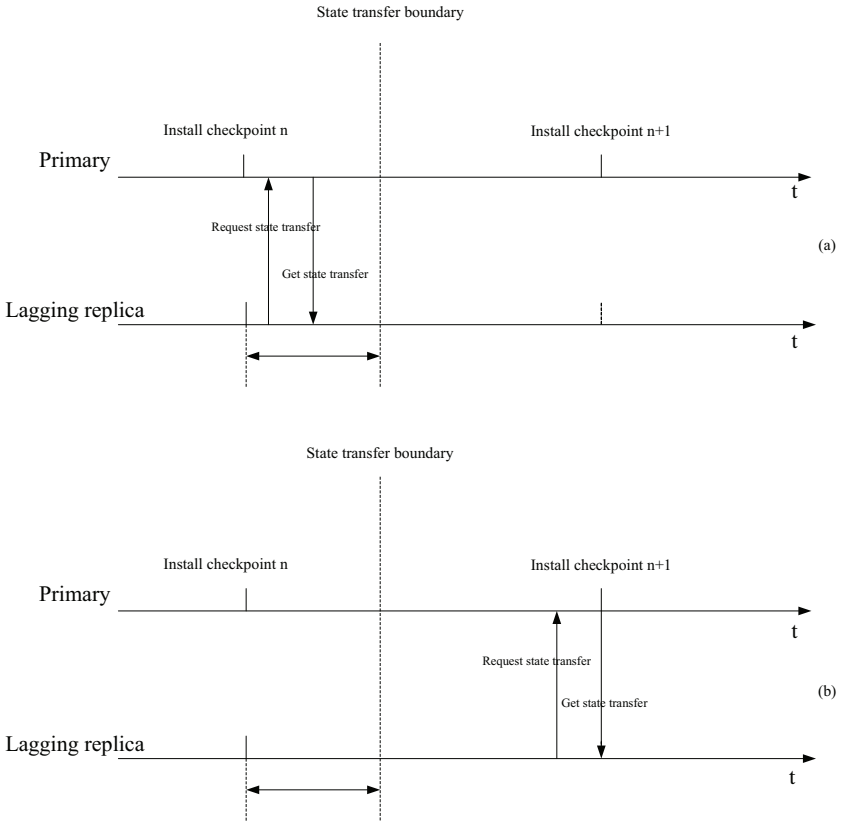


Figure 3: State transfer optimization.

CHAPTER V

PFT UNDER FIRE

5.1 Implementation

The PFT framework was implemented using the Java programming language and was built as a Java library for applications to use. The main components for the PFT framework are shown in Figure 4.

- *Multicast Sender(Client and Server)*: The Multicast Sender sends a message to the intended targets using UDP multicast.

- *Message Out Handler(Client)*: The Message Out Handler is for constructing outgoing messages for the client. The constructed messages are then past to the Multicast Sender to multicast to the server replicas.

- *Message Receiver(Client)*: The Message Receiver is for handling all incoming messages to the client. Upon receipt of a message, the Message Receiver carries out the following operations in sequence: (1) drops the message if it is duplicated or irrelevant, (2) dispatches the message to an appropriate handler based on the

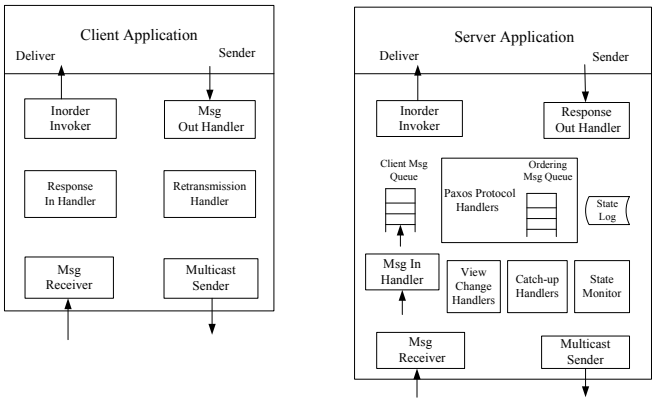


Figure 4: The main components of the PFT framework.

type of the message. If the message is a response to the previous request, it passes the response to the Response In Handler. Once the Response In Handler gets this message, it checks if it has received this message from majority of the server replicas. If it does, the handler can make sure the response is ready to be delivered to application and passes it to the Inorder Invoker component.

- *Message Receiver(Server)*: It is responsible to preprocess all incoming messages to the server side. The receiver processes more message types than the client side. Generally, those messages could be that client requests, accept or commit requests for ordering and accept responses. In exceptional cases, they could also be view change or view install requests, catch-up requests and responses, checkpoint messages, state transfer requests and responses. The receiver passes messages to different handlers according to their message types.

- *Message In Handler(Server)*: When the handler gets a new message, it first checks if the message is a duplicate. If the message is not a duplicate, it stores the message and notifies the Accept-request Out Handler (explained next) to prepare for ordering this message.

- *Paxos Protocol Handlers*: These handlers are responsible for total ordering client messages. They consist of Accept-Request In Handler, Accept-Request Out Handler, Accept Response In Handler, Accept Response Out Handler, Commit-Request In Handler and Commit-Request Out Hander. They get ordering messages from Message Receiver and process messages according to their message types. In each ordering phase, the to be ordered message is assigned a ordering number by the primary's Accept-request Out Handler. The handler constructs an accept request with the ordering number and gives it to the Multicast Sender. When another server's Accept-request In Handler gets the request from its Message In Handler, it accepts the message's order, picks the message from client request queue and puts it into the ordering queue. As the server accepts the message, its Accept-response Out Handler constructs an accept response message and give it to its sender. The Accept-response In Handler of the primary collects all accept responses. If the majority of the servers have accepted the message's order, the primary's Commit-request Out Handler constructs a commit request for all replicas. When other replicas' Commit-request In Handlers get this commit request, the order is committed and they notify their Inorder Invokers to deliver the ordered message.

- *Inorder Invoker*: It retrieves the ordered messages, and delivers them to the application in total order.

- *Response Out Handler(Server)*: The Response Out Handler is for constructing out response messages for the server replicas. The constructed messages are given to the Multicast Sender for multicasting the response messages to the clients.

- *View Change Handlers*: These handlers are in charge of coordinating the view

change process. They are composed of View Change Request In Handler, View Change Request Out Handler, View Change Commit In Handler and View Change Commit Out Handler. The View Change Request In Handler gets view change requests from its Message Receiver and the View Change Request Out Handler constructs the view change request. When the primary in the new view has collected view change requests from the majority server replicas, it installs a new view and its request out handler constructs a view install message to notify other server replicas.

- *State Log*: Each replica creates and maintains a state log in its local storage. This state log is used for each replica to record the ordering information and state information. A replica records the corresponding information whenever it makes an accept or commit decision on an application request. This implies that to order a request, two disk writes are involved. Because the random disk I/O is too slow, which would make the overhead of request ordering excessive. To improve the efficiency for accessing the state log, we write the state log in memory instead of on disk and flush the data to the disk asynchronously periodically. This optimization is implemented by using memory-mapped I/O provided by the Java nio library.

- *State Monitor*: It periodically checks if there is any missing information from its State Log. A server replica records information for each ordering in sequence. If the State Monitor finds a hole existing in the State Log or incomplete information for some orders, it invokes Catch-up handlers(will be discussed below) to fetch missing information back.

- *Catch-up Handlers*: They are used to request missing information from other server replicas. They are composed of Catch-up Request In Handler, Catch-up

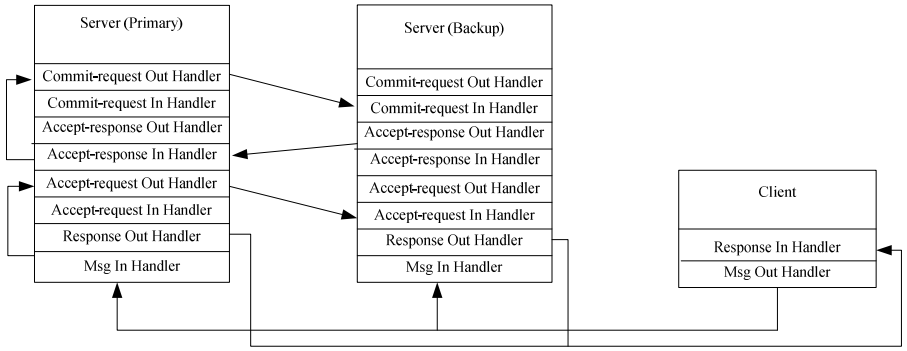


Figure 5: The interaction of the main components during normal operation.

Request Out Handler, Catch-up Response In Handler and Catch-up Response Out Handler. The Catch-up Out Handler constructs a catch-up request with missing information or missing orders. The Catch-up Request In Handler of the response server picks up the message and checks if it has the requested missing information or orders. It also decides if the requested server replica needs a state transfer. If it determines to respond, the Catch-up Request In Handler gives it to the Catch-up Response Out Handler. The Catch-up Response Out Handler constructs a response message with the needed information and gives it to its sender. The Message Receiver of the requester may get the response with or without a state-transfer and passes it to Catch-up Response In Handler to update its ordering information.

In the following, we describe how the components work together during the process of a request/response round trip during normal operation, as shown in Figure 5.

When application data is available. It is sent to the Message Out Handler. The handler constructs a message request with the data as the payload and passes it to the Multicast Sender. The sender multicasts the request to all server replicas.

Once the message receiver at a server replicas receives the message request from the client, it passes the message to the Message In Handler. If the handler has received this request before, it drops it; otherwise it stores the request in its client message queue and notifies the Accept-request Out Handler to handle this application message. If the replica is the primary, the Accept-request Out Handler assigns the newly arrived message the next available sequence number, puts it into the handler's ordering queue, and prepares the accept request with the assigned sequence number and other information pertinent to the application request being ordered. The out handler then passes the accept request to the Multicast Sender for sending to other replicas.

Once a backup's Accept-request In Handler gets the accept request, it checks the request queue to see if it has received and stored this application request being ordered. If not, it requests a retransmission of this message from the primary; otherwise it puts the message into its ordering queue for accepting this order if it has never accepted an order with higher ordering number than the current one. If the in handler can accept this order, it notifies the Accept-response In Handler to accept the request. The Accept-response Out Handler then constructs an accept response and gives it to the Multicast Sender.

The primary's Accept-response In Handler counts the responses collected from different replicas, including the one it would have sent. When it has collected responses from the majority of the replicas, the in handler notifies the Commit-request Out Handler to prepare a commit request and gives it to the Multicast Sender.

Once a backup's Commit-response In Handler gets the commit request, it retrieves the corresponding application request from its ordering queue and gives it to the Inorder Invoker component. The invoker then delivers the message to the server application.

When the server finishes process the request, the response is passed to the server's Response Out Handler. The handler constructs a response message and gives it to the Multicast Sender. The sender then sends the response to the client.

When the client receives the response message, it passes the message to its Response In Handler. The Response In Handler counts the responses from different server replicas, and when it has collected responses from the majority of the replicas, the in handler gives the response to its Inorder Invoker component. The invoker then delivers it to the client's application.

5.2 Experimental setup

All the experiments are carried out on a local area network with 12 Dell SC440 servers connected by a 100 Mbps Ethernet switch. Each server is equipped with a single Pentium D 2.8 GHz processors and 1 GB memory running SUSE 10.2 Linux.

A simple client-server application is used in our experiments. The server is replicated across a number of server nodes. Each client sends requests to all server replicas using UDP multicast. For each run, a client issues at least 1000 requests without any think time consecutively. One server replica is designated as the primary, which performs tasks of the unique proposer and also acts as both an acceptor and a learner. The primary is in charge of leading the ordering process for application requests. The remaining replicas are backups, playing the roles of acceptor and learner. In the experiments, we vary the number of replicas with an upper-bound on the replication degree of 9.

The PFT framework was evaluated by a wide variety of experiments under various network conditions and different workloads (e.g., request sizes and request bursts). Each experiment focuses on evaluating one aspect of the replication protocol design, such as batching, catch up and view change mechanisms.

5.3 Basic evaluation

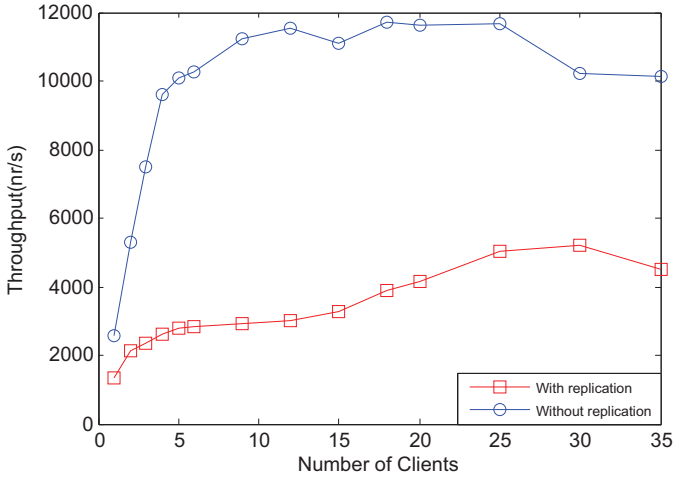
5.3.1 Procedure

For basic performance evaluation, we assess the runtime overhead of our PFT framework during normal operation. The runtime overhead is determined by comparing the performance of the test application with our PFT framework and that without replication under the same workload and network condition. The system with replication is composed of 3 replicas and various number of clients. In this experiment, the varying load is controlled by running different number of concurrent clients. We vary the number of clients from 1 to 30. Each client, once started, continuously issues requests to the server without think time. Under each workload, we measure the end-to-end latency at the client and the system throughput at the server.

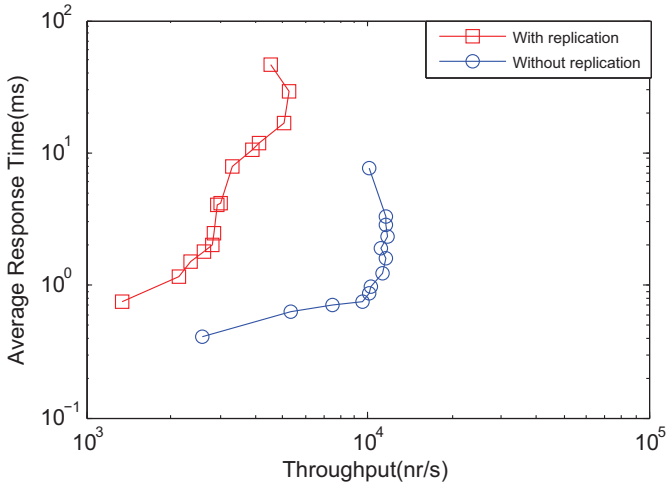
5.3.2 Results

The experimental results are summarized in Figure 6. Figure 6(a) shows the throughput measurement results with and without replication. In both measurements, the throughput increases quickly when more concurrent clients are launched. However, without replication, the system reaches the maximum throughput when the number of concurrent clients is increased to about 10. Whereas when replication is used, it takes a lot more clients for the system to reach maximum throughput. This is expected because the system is side-tracked with the total ordering activities when replication is used, which involves with multiple waiting-for-message operations.

The end-to-end latency as a function of system throughput for both measurements is reported in Figure 6(b). As can be seen in Figure 6(b), when the load exerted on to the server is close to its capacity, the latency quickly rises. When the load keeps increasing, the throughput degrades. With replication, due to the cost of total



(a) The throughput distribution vs the number of clients for the replication and non-replication system.



(b) Latency vs throughput curves for the replication and non-replication systems.

Figure 6: Performance comparison for the replication system and non-replication system.

ordering, the maximum throughput is about half of that of a non-replicated system.

5.4 Batching

5.4.1 Procedure

In this section, we report the performance of the PFT framework with the batching mechanism. First, we compare system performance with and without batching, then we evaluate the system performance with batching under various conditions: different batch sizes, different number of replicas and various workload. Accordingly, the experiment is carried out in two steps. In the first step, we redo the measurements on throughput and end-to-end latency under the same conditions as those without batching. In the second step, a comprehensive evaluation is conducted under the following conditions:

(1) Varying the batch size

We measure throughput under various batch sizes (from 1 to 40 requests) while keeping the number of replicas fixed. A batching timer with 100 ms timeout value is initiated at the primary. The primary keeps collecting requests from clients until either the batching timer expires or there are enough number of requests queued at the batching buffer. When it is ready to order a batch, the primary resets the batching timer and initiates the total ordering process. We use two different load conditions: with 30 concurrent clients, and the other with 50 concurrent clients. To accurately capture the dynamics of the system with batching, we take periodic sampling of the throughput (once for every 100 requests processed) at the replicas, and plot the distribution of the sampled throughput in terms of probability density function (referred to as PDF in short) of the throughput.

(2) Varying the number of replicas

We assess the fault scalability of the PFT framework with batching. For each run, we keep the batch size fixed, and vary the number of replicas such that we can tolerate different number of faults f (i.e., the total number of replicas needed to tolerate f faults is $2f + 1$). For comparison, we repeat the experiment on the base PFT framework (without batching). We use 3 to 9 replicas to tolerate 1 to 4 faults.

(3) Varying workload

The objective of this experiment is to evaluate the system performance with batching under different workloads. The different workloads are produced by changing the request size (from 2 to 1024 bytes) and the number of concurrent clients (from 1 to 60).

5.4.2 Results

From Figure 7, it can be observed that batching can greatly improve the throughput of the PFT framework, and reduce the latency under the same throughput. The performance enhancement is more significant when the load is high (i.e., when the number of clients is large), which is instrumental in achieving better scalability.

Figure 8(a) shows the throughput distribution of the system with different batch sizes. As can be seen in Figure 8(a), the peak throughput increases dramatically as the batch size is increased from 1 to 5. As we keep increasing the batch size from 15 to 30, the peak throughput is increased with less probability density and the distribution becomes less converged. Table I summarizes the average throughput and latency for the system configured with different batch sizes. As can be seen from the table, when the batch size increases, the throughput is improved and the end-to-end latency is reduced.

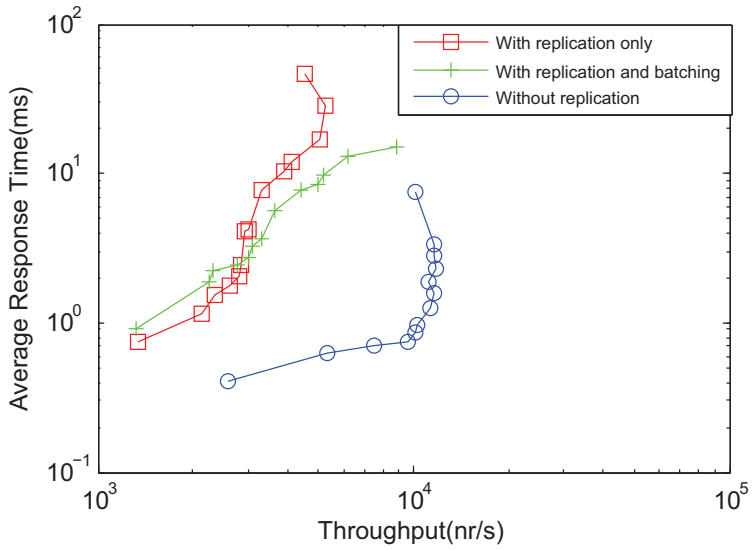


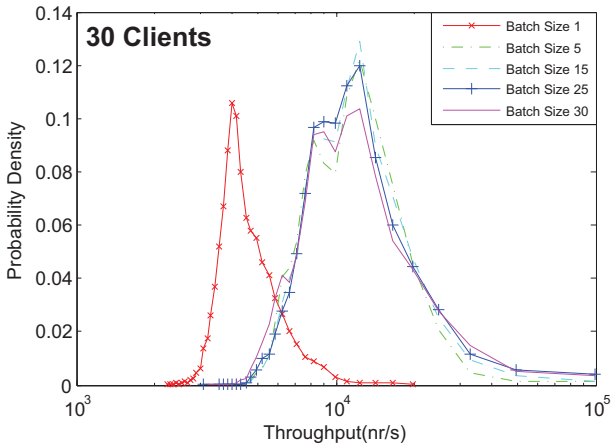
Figure 7: End-to-end latency as a function of system throughput for different batch sizes (with 30 concurrent clients).

Batch size	Average throughput(ops/s)	Average latency(ms)
1	5933	28.2
5	6457	16.1
15	6369	14.9
25	6584	14.6
30	6348	14.5

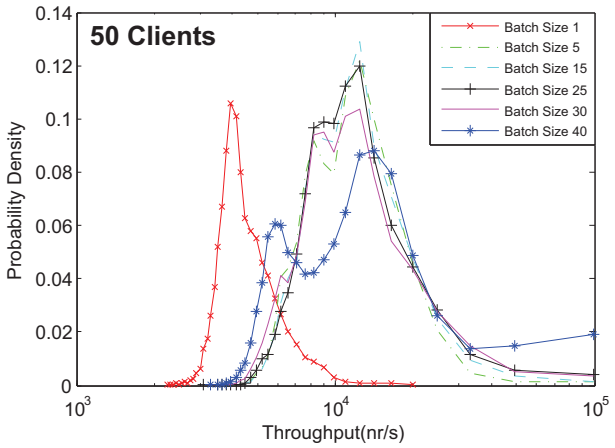
Table I: Average throughput and latency under batching(a).

The throughput distribution with 50 concurrent clients is similar, as shown in Figure 8(b). The peak throughput increases dramatically as the batch size is changed from 1 to 5. As we keep increasing the batch size from 25 to 40, the peak throughput is increased with less probability density and the distribution becomes less converged. Table II lists the average throughput and latency for the system configured with different batch sizes.

Even though both and peak throughput and average throughput are significantly better with batching is enabled, the throughput distribution measurement (as shown in Figures 8(a) and 8(b)) reveals that there exist many samples with low throughput



(a) Throughput distribution under different batch sizes (with 30 concurrent clients).



(b) Throughput distribution under different batch sizes (with 50 concurrent clients).

Figure 8: Throughput distribution under different batch sizes.

Batch size	Average throughput(ops/s)	Average latency(ms)
1	4509	46.5
5	9900	44.8
15	11494	43.5
25	11780	32.7
30	11754	30.5
40	12399	28.8

Table II: Average throughput and latency with batching(b).

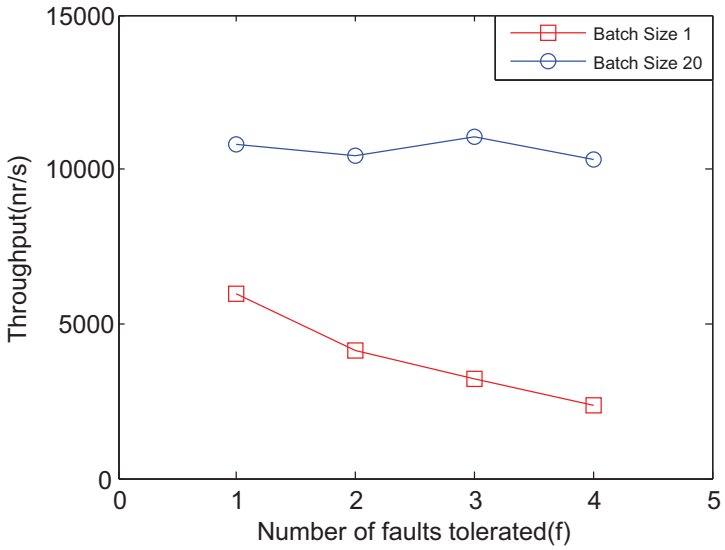
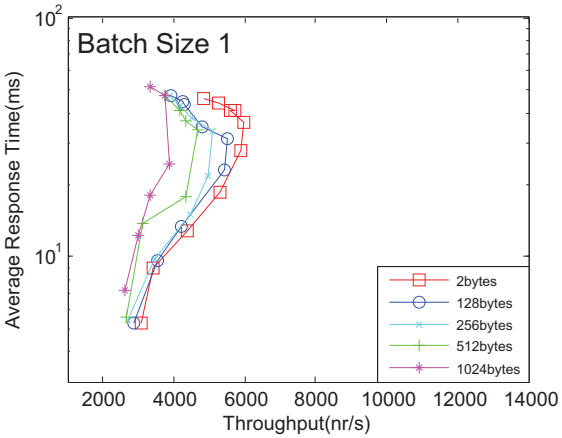


Figure 9: Fault scalability with and without batching.

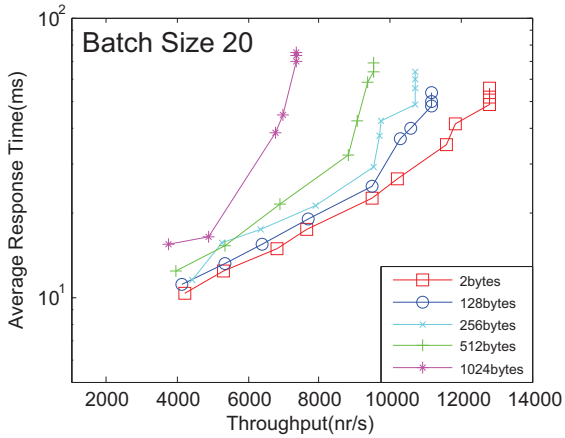
values. This is not surprising because the actual batch size is not always the maximum batch size we set. Due to the asynchrony of the system, if the previous batch of requests are not processed in time, the clients would not submit new requests (a client issues requests synchronously), and hence, the actual batch size could be very low for the new batch.

Figure 9 shows the system throughput with different number of faults tolerated, with and without batching. It can be observed that the batching mechanism has significantly increased the fault scalability. Without batching, the system throughput degrades apparently as the number of replicas increases. It is because increasing the replication degree will introduce more load to the system (both heavier communication cost and CPU processing overhead). However, with batching, the system throughput is dropped only slightly when the number of replicas increases.

Figure 10(a) and Figure 10(b) show the measurement results with various request sizes. Figure 10(a) shows the throughput results without batching (i.e., when the



(a) Throughput as a function of request sizes (with the batch size of 1).



(b) Throughput as a function of request sizes (with the batch size of 20).

Figure 10: Throughput as a function of request sizes.

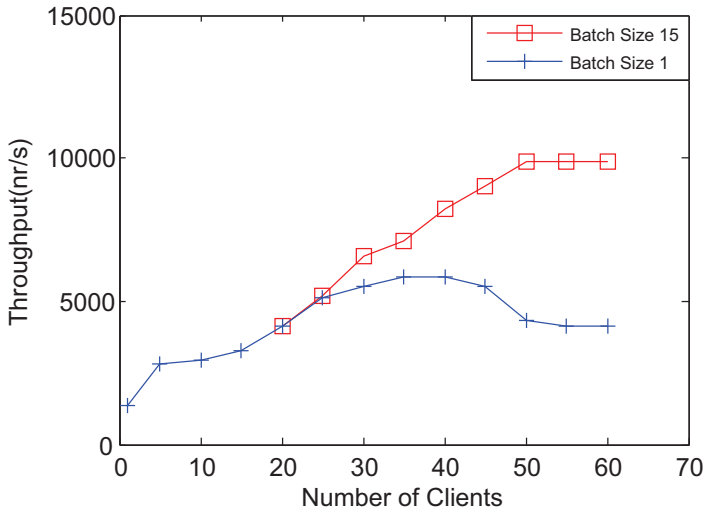


Figure 11: Throughput as a function of number of concurrent clients.

batch size is set to 1). Figure 10(b) shows the throughput results with a batch size of 20 under the same workload. In both cases, the throughput (in terms of the number requests serviced per second) degrades as the request size increases. However, the throughput degradation when batching is enabled is less severe. It again demonstrates that batching improves the system scalability.

Next, we report the experimental results (with and without batching) obtained by varying the number of concurrent clients. In Figure 11, as too many clients are involved to produce load, the throughput without batching becomes dropping. However, the peak throughput with batching is almost twice more than that without batching due to the reduced overhead of message total ordering. It can be seen that batching can improve the efficiency of total ordering.

The experimental results confirmed that batching indeed helps to achieve better performance by amortizing the cost of total ordering over a batch of requests. The throughput under heavy load is significantly increased, and the end-to-end latency

is reduced as well. As a result, both the fault scalability and load scalability are improved.

5.5 Catch-up mechanism

5.5.1 Procedure

In this section, we study experimentally the effectiveness of two catch-up mechanisms, namely, MBQ and PBQ. In the experiments, we first measure the time interval when two consecutive message requests arrive at a replica without message loss. In later text, we refer to the time interval as the message pick-up interval. The experiments aim to capture the dynamic workload conditions on the replica for the two catch-up mechanisms. The distribution of the interval can reflect the workload variation in the network environment and help us to evaluate the two catch-up mechanisms. In addition, we profile other important factors on the system performance, such as throughput and end-to-end latency. The workload is produced by running different numbers of concurrent clients (ranging from 1 to 10). For each run, every client issues 10,000 concurrent message requests to the replicated server (with replication degree of 5 and 7).

To study the effectiveness of the catch-up mechanisms, we have to induce the overload situations in a controlled manner and without any catch-up mechanism configured. When a replica is overloaded, it may start losing messages (due to input buffer overflow). We create temporary overload at one of the replicas by creating heavy CPU load for several times. Each time we keep the load running for nearly 5 s until the replica has message loss, then we remove the load from the replica until it stops losing messages. After study the message loss scenarios, we run the replication system with the two catch-up mechanisms separately. We create temporary overload

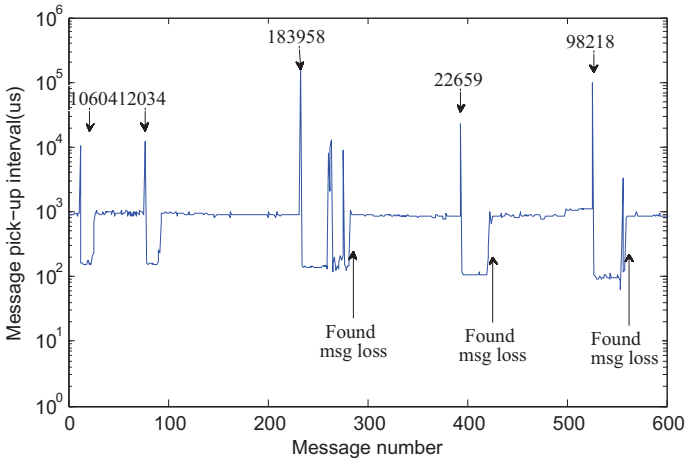
at a replica and observe the message loss level for both mechanisms.

For the MBQ mechanism, the replicas periodically exchange ordering information. A timer is used by each replica for this purpose. The timeout value used is initiated to 100 ms. The timeout value is chosen this way so that it is not too short which could lead to too many exchange messages, and yet it is not too large which could render the mechanism ineffective.

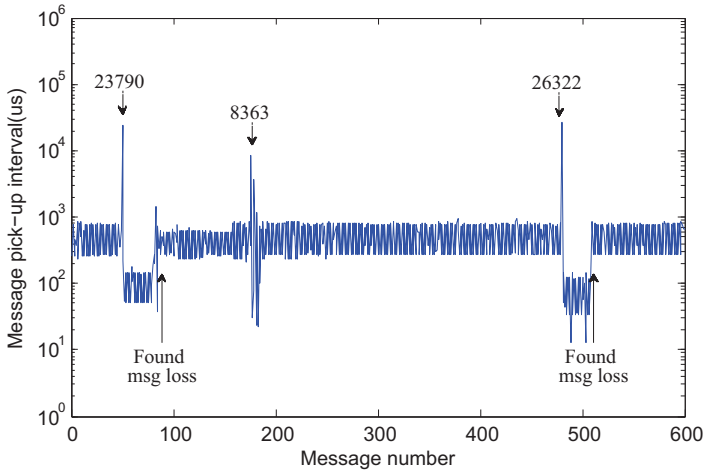
5.5.2 Results

The distribution of the message pick-up interval without any catching up mechanism is shown in Figure 12(a) and Figure 12(b). As can be seen from the experimental results, the average message pick-up interval is 0.85 ms in the presence of a single client, and the interval is shortened to 0.73 ms with two concurrent clients. Under normal load condition, the interval is fairly constant. However, when message loss occurs, the message pick-up interval is much larger than the normal value. As shown in Figure 12(a), the message pick-up interval reaches to 183.958 ms, 22.659 ms and 98.218 ms at three message loss occasions. Similarly, as shown in Figure 12(b), the message pick-up interval reaches to 23.796 ms and 26.322 ms when message losses occur.

It is apparent that the large delay in picking up the next request by a replica is caused by temporary overload (i.e., the replica was busy doing something else). In the worse case, a message loss would occur due to buffer overflow. From the figures, it also can be observed that after the overload is removed, the replica resumes picking up messages from its buffer and the interval becomes much shorter than the average value. This is expected because when the overload is removed, there are already a batch of requests queued up at the buffer and they can get picked up immediately. There is a gap between the normal value and the lowest value of the message pick-up



(a) Message lost scenario as one client is involved.



(b) Message lost scenario as two clients are involved.

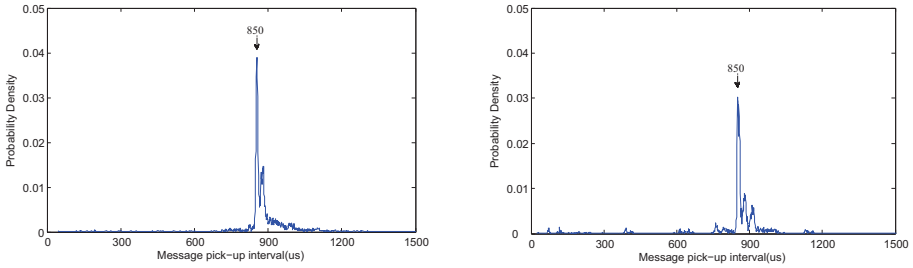
Figure 12: Message lost scenario with different number of clients involved.

interval. It is because under limited number of clients the round-trip time needs the replica waiting for the message arrival. If there are more clients involved in sending messages, the average gap will become smaller.

The effectiveness of the two catch-up mechanisms on handling overload situations is first evaluated by profiling the distribution of the message pick-up intervals. For this experiment, only a single client is involved to send message request. Figure 13(a)-(c) shows the probability density function of the pick-up interval for normal load, overload under MBQ, and overload under PBQ, respectively. Figure 13(d) shows the distribution of the message pick-up interval without any catch-up mechanism configured. From this figure, it can be observed that the replica encounters serious overload and has much worse performance than the other three cases. Figure 13(a) shows that the message pick-up interval under the normal load is narrowly converged at 850 μs. From Figure 13(b), it can be observed that the distribution for PBQ is very close to that for normal load, except for some small disturbance. The distribution for MBQ (shown in Figure 13(c)) is much different. The distribution of the message pick-up interval for MBQ is less converged and reflects larger variance. It is probably due to the frequent ordering information exchanges the MBQ introduces. Under the condition of this experiment (short state size), it appears that PBQ is more favorable than MBQ to handle temporary overload conditions.

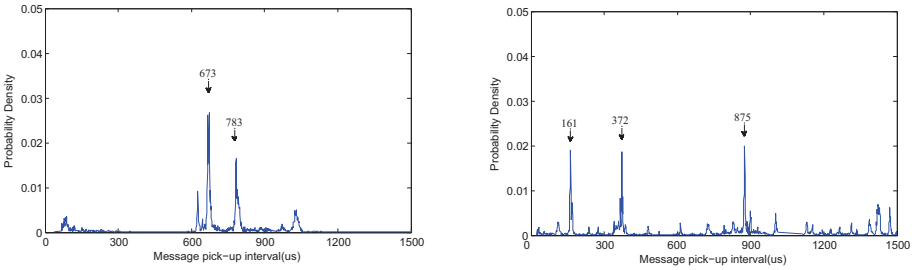
We further measure the throughput and latency of the system for both mechanisms under similar controlled overload conditions. In addition, we profile the following important statistics for each of the mechanisms:

- For each run, how many times an overloaded replica has to initiate the catching up mechanism;
- How long it takes to complete the catching up operation;
- How many state transfers are involved in each run;



(a) Message pick-up interval under no message loss behavior.

(b) Message pick-up interval for PBQ under message loss behavior.



(c) Message pick-up interval for MBQ under message loss behavior.

(d) Message pick-up interval for no catch-up strategy configured under message loss behavior.

Figure 13: Performance comparison with using message pick-up interval.

- How many replicas lose messages in each run.

	MBQ	PBQ	None
Lagging replicas	4	2	3
Catch-up occurring counts	6	10	0
Catch-up lasting time(ms)	3	16	0
State transfers	2	1	20
Throughput(ops/s)	1955	1595	731
Latency(ms)	4.3	4.6	7.1

Table III: Performance for MBQ and PBQ with 5 replicas involved.

	MBQ	PBQ	None
Lagging replicas	6	3	4
Catch-up occurring counts	6	8	0
Catch-up lasting time(ms)	6	16	0
State transfers	5	2	20
Throughput(ops/s)	1388	1250	568
Latency(ms)	4.0	4.2	6.3

Table IV: Performance for MBQ and PBQ with 7 replicas involved.

Table III and Table IV show the performance for both mechanisms in terms of the throughput and latency, the time it takes for a replica to catch up with other replicas, the occurrence of the catch-up operations and the number of state transfers with different replication sizes. From both tables, it can be observed that the time it takes for a replica to catch up with other replicas using MBQ is shorter than that for PBQ. Similarly, the throughput is higher when MBQ is used. Most importantly, the performance of the system is much worse if no catch-up mechanism is used.

However, when MBQ is used, more replicas suffer from message losses (with a ratio about 2 to 1) than that if PBQ is used. Similarly, more state transfers are observed when MBQ is used. It is probably due to MBQ's use of frequent multicasting among replicas that cause the message loss propagate to other replicas. However, from both tables, the throughput and latency for MBQ are both better than those for PBQ,

possibly because in many cases the lagging replicas for MBQ can pick up messages faster than those for PBQ. According to the previous experiment, the message pick-up interval becomes much shorter than normal when overload occurs and MBQ is easier to have this scenario. If extreme overload lasts for long time, both mechanisms could meet countless catching ups. In this situation, the catch-up mechanisms may need to be removed.

In conclusion, through the experiment, it can be concluded that both catch-up mechanisms can be used under temporary overload conditions. However, MBQ performs better than PBQ due to higher throughput and lower latency. In this experiment, we use small state size for our system. If the state size is large, we think PBQ which has less state transfers may perform better.

5.6 View change

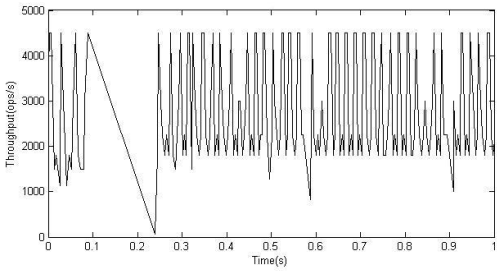
5.6.1 Procedure

The timeout value used in the view change timer has to be carefully determined. If it is set too short, unnecessary view changes will be triggered by moderate temporary overload at the primary. If it is set too large, it may take too long for reconfiguration to take place, which reduce the system responsiveness. In the PFT framework, the initial timeout value is set to 100 ms, the value is subsequently dynamically adjusted during runtime based on the observed total ordering latency. The performance of our view change mechanism is evaluated using our test application with 5 server replicas. Each time the primary crashes, any backup can trigger the view change and a new primary is chosen in the new view. The recovery time (since the primary is crashed until a new view is installed) and the view change latency (i.e., the time it takes for the view change to complete) are measured at each backup replica. Furthermore,

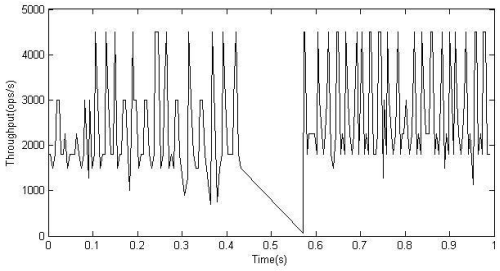
the time it takes to handle the accept records during the view change is profiled at each replica. Finally, the impact of the batching on the view change latency is also studied.

5.6.2 Results

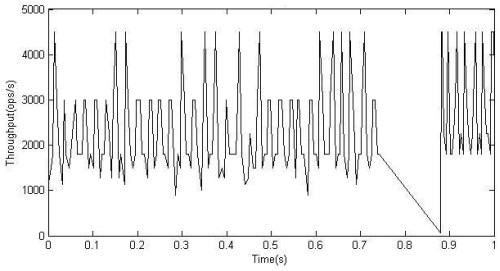
Figure 14 shows the observed throughput values at the four backup replicas around a triggered view change. When the primary is crashed, the throughput drops significantly and when the view change completes, the throughput rise back to normal value. Therefore, we can observe the total recovery time and the view change latency by taking frequent samples of the throughput at each replica. As shown in Figure 14(a), the primary ($Replica_0$) is crashed at about 0.10 s. $Replica_1$'s view change timer expires at 0.23 s. The actual elapsed time since the view timer was started is measured to be 118 ms (this is the fault detection latency). $Replica_1$ initiates the view change by sending out a view change request. After collecting two view change requests from other non-fault replicas, $Replica_1$ installs a new view and becomes the new primary at about 0.24 s. It takes $Replica_1$ almost 7 ms to complete the remaining view change process. The view change latency as observed by $Replica_1$ is $7 + 2.4 = 9.4$ ms. The total recovery time is the sum of the fault detection time and the view change latency, which is $118 + 9.4 = 127.4$ ms. The recovery time can be also estimated at the client by comparing the end-to-end latency during the view change and that during normal operation. Under normal operation, the average end-to-end latency is 2.25 ms. However, the latency rises to 130 ms due to the view change. The extra delay can be attributed to the failure recovery, which is about $130 - 2.25 = 127.75$ ms. This is consistent with the throughput-based measurement as shown in Figure 14(a). The recovery time and view change latency as measured by the other three backup replicas can be similarly determined by Figures 14(b),(c), and(d).



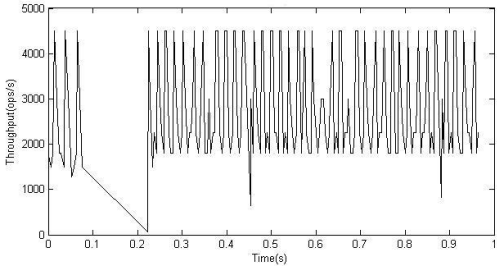
(a) Throughput vs. elapsed time around the view change as observed by Replica 1.



(b) Throughput vs. elapsed time around the view change as observed by Replica 2.



(c) Throughput vs. elapsed time around the view change as observed by Replica 3.



(d) Throughput vs. elapsed time around the view change as observed by Replica 4.

Figure 14: Throughput vs. elapsed time around the view change.

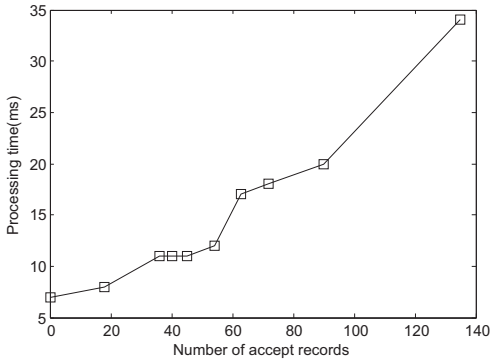
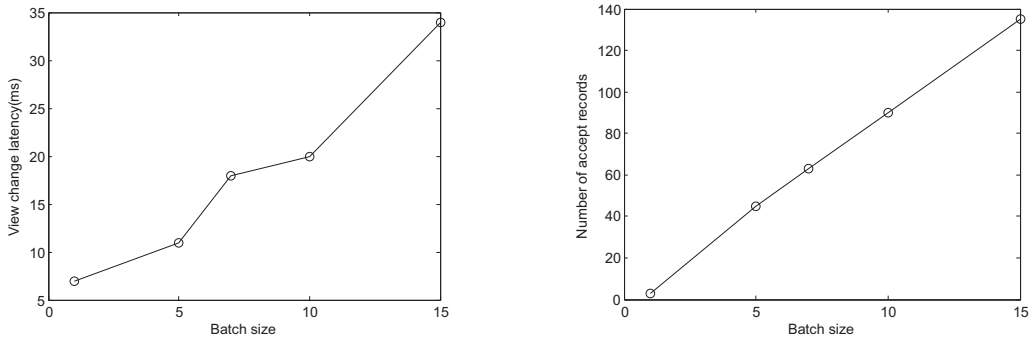


Figure 15: View change latency as a function of the number of accept records.



(a) View change latency versus the batch size. (b) Number of accept records versus the batch size.

Figure 16: The impact of the batch size on the view change performance.

We further investigate the impact of the number of accept records included in the new view message on the view change latency. The view change latency as a function is shown in Figure 15. As can be seen, the view change latency can be increased significantly when the number of accept records involved are large.

Because the larger batch size is used, the more accept records will be involved during the view change. It is not surprising to see a similar dependency of the view change latency, as shown in Figure 16.

CHAPTER VI

SUMMARY AND FUTURE RESEARCH

6.1 Conclusion

In this thesis, a lightweight fault tolerance framework (referred as PFT framework) is presented. This framework is adapted from the Paxos algorithm with consideration of practical issues such as good performance under normal operation and fast recovery. A comprehensive performance evaluation for the PFT framework is also presented. In particular, the effectiveness of various optimization mechanisms we introduced to the PFT framework is assessed. In addition, the performance of the view change mechanism is studied. The experiments show that PFT framework exhibit optimal performance and is robust under various network and load conditions.

6.2 Future Work

In the future, we plan to investigate how to adapt our PFT framework for use in wide-area networks. Such a framework would be useful to tolerate total site-failures.

We are also interested in exploring the use of PFT framework in high-availability clusters for fault tolerant services such as totally ordered reliable multicast service, health monitoring service, and membership service. Furthermore, we are planning to open-source our PFT implementation, hoping to benefit the peers who are interested in building highly available systems.

BIBLIOGRAPHY

[1] M. Castro, B. Liskov, "Practical byzantine fault tolerance," *in Proceedings of the Third ACM Symposium on Operating Systems Design and Implementation*, pp. 173-186, 1999

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, Vol. 16(2), pp. 133-169, 1998

[3] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," *in Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 335-350, 2006

[4] B. W. Lampson, "How to build a highly available system using consensus," *in Distributed Algorithms, ed. Babaoglu and Marzullo, Lecture Notes in Computer Science 1151, Springer*, pp. 1-17, 1996

[5] T. D. Chandra, R. Griesemer, J. Redstone, "Paxos Made Live - An Engineering Perspective," *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*, 2007

[6] J. P. Martin, L. Alvisi, "Fast Byzantine Paxos," *Technical Report TR-04-07*, 2004

[7] Y. Amir, J. Kirsch, "Paxos for system builders," *in LADIS '08: Proceedings of Large-Scale Distributed Systems and Middleware*, New York, September 2008

[8] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column)*, Vol. 32(4), pp. 18-25, Dec. 2001

[9] F. Junqueira, Y. Mao, K. Marzullo, "Classic Paxos vs. Fast Paxos: Caveat Emptor," *in Proceedings of USENIX Hot Topics in System Dependability (HotDep)*, 2007

[10] M. J. Fischer, "The consensus problem in unreliable distributed systems," 1983

[11] Y. Song, V. Renesse, R. Schneider, "Evolution vs. Intelligent Design in Consensus Protocols," 2007

[12] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. Reiter, J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," *in Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005

[13] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, E. Wong, "Zyzzyva, Speculative Byzantine Fault Tolerance," *in Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2007

[14] L. Lamport, "Fast Paxos," *Distributed Computing*, Vol. 19(2), pp. 79-103, 2006

[15] L. Lamport, M. Mike, "Cheap Paxos," *in Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2004

[16] L. Lamport, "Generalized Consensus and Paxos,"
http://research.microsoft.com/users/lamport/pubs/pubs.html, 2005

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, Vol. 21(7), pp. 558-565, 1978

[18] F. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: A tutorial," *Computing Surveys*, Vol. 22(3), pp. 299-319, 1990

[19] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems," *Computer Networks*, 1978

[20] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, L. Shrira, "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance," *in Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 2006

[21] L. Camargos, R. Schmidt, F. Pedone, "Multicoordinated Paxos," 2006

[22] E. Gafni, L. Lamport, "Disk Paxos," *International Symposium on Distributed Computing*, pp. 330-344, 2000

[23] A. Sing, T. Das, P. Maniatis, P. Druschel, T. Roscoe, "Bft protocols under fire," *in NSDI*, 2008

[24] W. Zhao, "A Lightweight Fault Tolerance Framework for Web Services," *IEEE/WIC/ACM International Conference on Web Intelligence (WI'07)*, pp. 542-548, 2007