2014

Byzantine Fault Tolerance for Distributed Systems

Honglei Zhang
Cleveland State University

BYZANTINE FAULT TOLERANCE FOR DISTRIBUTED SYSTEMS

HONGLEI ZHANG

Master of Science in Electrical Engineering

Cleveland State University

Dec 2007

submitted in partial fulfillment of requirements for the degree

DOCTOR OF ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

April, 2014

We hereby approve the dissertation
of
Honglei Zhang

Candidate for the Doctor of Engineering degree.
This dissertation has been approved for the Department of

Electrical and Computer Engineering

and CLEVELAND STATE UNIVERSITY
College of Graduate Studies by

_____

Wenbing Zhao, Dissertation Committee Chairperson – Department & Date

_____

Nigamanth Sridhar, Dissertation Committee Member – Department & Date

_____

Yongjian Fu, Dissertation Committee Member – Department & Date

_____

Lili Dong, Dissertation Committee Member – Department & Date

_____

Janche Sang, Dissertation Committee Member – Department & Date

_____

Student's Date of Defense

This student has fulfilled all requirements for the Doctor of Engineering degree.

_____

Dan Simon, Doctoral Program Director

To my beloved wife Shuqiong...

ACKNOWLEDGEMENTS

Many thanks go to my advisor, Dr. Wenbing Zhao, for making me do this research, for all his abundantly help and full-of-insight support, for guidance as my supervisor, and for the virtues I learned from him.

Express my gratitude is also due to Dr. Lili Dong, Dr. Yongjian Fu, Dr. Nigamanth Sridhar, and Dr. Janche Sang for agreeing to be on my committee and for all of their assistance.

Special thanks to Dr. Ye Zhu, Dr. Pong P. Chu, Dr. William L. Schultz and Dr. Dan Simon for their elaborately prepared lectures which provide very helpful knowledge to enhance my view.

Thanks to all my friends for their kind help and friendship, and also for sharing the literature and invaluable assistance.

I would also thank my wife, my daughter, my mom and all my relatives. They stand by me and support me all the time.

BYZANTINE FAULT TOLERANCE FOR DISTRIBUTED SYSTEMS

HONGLEI ZHANG

ABSTRACT

The growing reliance on online services imposes a high dependability requirement on the computer systems that provide these services. Byzantine fault tolerance (BFT) is a promising technology to solidify such systems for the much needed high dependability. BFT employs redundant copies of the servers and ensures that a replicated system continues providing correct services despite the attacks on a small portion of the system. In this dissertation research, I developed novel algorithms and mechanisms to control various types of application nondeterminism and to ensure the long-term reliability of BFT systems via a migration-based proactive recovery scheme. I also investigated a new approach to significantly improve the overall system throughput by enabling concurrent processing using Software Transactional Memory (STM). Controlling application nondeterminism is essential to achieve strong replica consistency because the BFT technology is based on state-machine replication, which requires deterministic operation of each replica. Proactive recovery is necessary to ensure that the fundamental assumption of using the BFT technology is not violated over long term, i.e., less than one-third of replicas remain correct. Without proactive recovery, more and more replicas will be compromised under continuously attacks, which would render BFT

ineffective. STM based concurrent processing maximized the system throughput by utilizing the power of multi-core CPUs while preserving strong replication consistency.

TABLE OF CONTENTS

Page

ACRONYMS

BFT: Byzantine Fault Tolerance

PBFT: Practical Byzantine Fault Tolerance

STM: Software Transactional Memory

Q/U: Query/Update

HQ: Hybrid Quorum

ND: Nondeterminism

VPRE: Verifiable pre-determinable nondeterminism

NPRE: Non-verifiable pre-determinable nondeterminism

VPOST: Verifiable post-determinable nondeterminism

NPOST: Non-verifiable post-determinable nondeterminism

VNPRE: Composite type with both VPRE and NPRE

VPRE-NPOST: Composite type with both VPRE and NPOST

NPRE-NPOST: Composite type with both NPRE and NPOST

VNPRE-NPOST: Composite type with VPRE, NPRE, and NPOST

ndet: Nondeterminism value

postnd: Post nondeterministic value

C-BFT: Concurrent Byzantine Fault Tolerance

S-BFT: Sequential Byzantine Fault Tolerance

T_i: Transaction with sequence number i

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER I

INTRODUCTION

In today's society, Internet has become an irreplaceable part in people's life and online services are playing a more and more important role. Naturally, the services are expected to be highly available despite arbitrary faults (referred to as Byzantine faults [34]). To achieve high availability, the system should always be ready to provide correct services to its clients even if a small portion becomes Byzantine faulty.

Byzantine fault tolerance (BFT) is a promising state-machine based replication technique. However, existing BFT algorithms, proposed so far in [13, 14, 15, 16, 20, 68], can only deal with applications with deterministic operations or those with the simplest types of replica nondeterminism. To handle replica nondeterminism found in practical applications, the BFT algorithm has to be improved in order to prevent the system from being exploited due to the presence of replica nondeterminism. In chapter 3, we introduce a set of mechanisms to control different types of nondeterminism for Byzantine fault tolerance.

BFT algorithms assume that less than one-third of the replicas can be faulty. Without additional mechanisms, this assumption could not hold over long-run because adversaries would continue trying to compromise more and more replicas. To prevent this from happening, various proactive recovery schemes [13, 15, 48, 53, 54, 55] for BFT have been proposed. Common to all such schemes, the replicas are proactively restarted before it is known that they have become faulty. After analyzing the existing proactive recovery schemes, we noticed three issues. First, rebooting with a refreshed state may not be effective in repairing a replica if there are hardware damages. Second, even if a compromised replica can be repaired by rebooting, it usually is a prolonged process, which may cause the system to be unavailable during the recovery period. Third, all active replicas need to coordinate such that only a small portion of the replicas are undergoing recovery at any given time to ensure the completion of the recovery. In chapter 4, we present an alternative way for proactive recovery based on service migration. Our objective is to provide proactive recovery for long-running BFT systems while effectively controlling of all three issues.

With the combination of BFT and proactive recovery, distributed applications can be made more trustworthy. However, in existing BFT algorithms, all application requests have to be executed sequentially to ensure strong replica consistency. This inevitably imposes a severe limitation on the performance of BFT systems. In particular, they cannot fully exploit the power of modern multi-core processors which is pervasively available today. This issue has been addressed by a number of researchers [28, 19, 17, 60]. This limitation can be lifted by enabling concurrent execution by incorporating the software transactional memory (STM) technique into BFT systems. By using the

software transactional memory model for processing, it is possible to delivery multiple requests for concurrent execution as long as the commit order is controlled such that the order conforms to the total ordering of the requests that triggered the transactions. The software transactional memory technique can be further used to work with the speculative BFT algorithm [68]. With commit barrier and multi-version support, a request is delivered for speculative processing even if there are some conflicts. While most of the time the speculation works, it would require an abort and restart if the speculation is wrong. In chapter 6, we present our STM based speculative concurrent BFT framework which significantly improved the overall performance.

Finally, the conclusion and future work are described in chapter 7.

CHAPTER II

BACKGROUND

This chapter provides an overview of a number of topics that this dissertation research has been involved with, including Byzantine fault tolerance (BFT), strong replica consistency, software transitional memory (STM), concurrent and speculative execution, and concurrency control.

2.1 Byzantine Fault Tolerance

2.1.1 Byzantine Fault

The term "Byzantine fault" was coined by Lamport [34] as part of the classic coordination problem known as the Byzantine Generals Problems. A Byzantine fault refers to an arbitrary fault that may occur during the execution of a distributed system which may lead the system to an arbitrary failure state. A Byzantine fault may make the

system respond to a client's request in an unpredictable way, such as crash, executing incorrect instructions, or processing the right instructions with wrong order. Compared with fail-stop faults, Byzantine faults are much harder to detect and, even worse, Byzantine faulty components may collude together, which could make the detection of such faults much harder.

2.1.2 Byzantine Fault Tolerance

Byzantine fault tolerance refers to the capability of a system to tolerate Byzantine faults. It can be achieved by replicating the service and ensuring all service replicas to reach Byzantine agreement on all state transitions. Byzantine agreement refers to the procedure that ensures all correct components reaching a consensus despite the presence of Byzantine failures.

The first highly efficient Byzantine agreement algorithm was introduced by Castro and Liskov [13, 14] (referred to as the BFT algorithm or Byzantine Agreement). This algorithm requires at least $3f + 1$ replicas to tolerate up to f Byzantine faulty replicas (f refers to the number of faulty nodes). During anytime of the execution, one replica is designated as the primary while the rests are played as backups. The BFT algorithm includes two modes of operations: normal operation, used to reach Byzantine agreement, and view change, used to handle the primary failures.

The normal operation involves three phases executed sequentially, followed by the execution order, they are called pre-prepare phase, prepare phase, and commit phase. In the pre-prepare phase, the primary multicasts a pre-prepare message to all backups as

its proposal. If a backup accepts the message, it starts the second phase, i.e. prepare phase, by multicasting a prepare message. When a replica has collected $2f$ matching prepare messages from different replicas, it concludes the prepare phase. Then the replica goes into the commit phase by multicasting a commit message to other replicas. The third phase ends when a replica has received $2f + 1$ matching commit messages from different replicas. Figure 1 shows the details of BFT algorithm in the normal operations with $f = 1$.

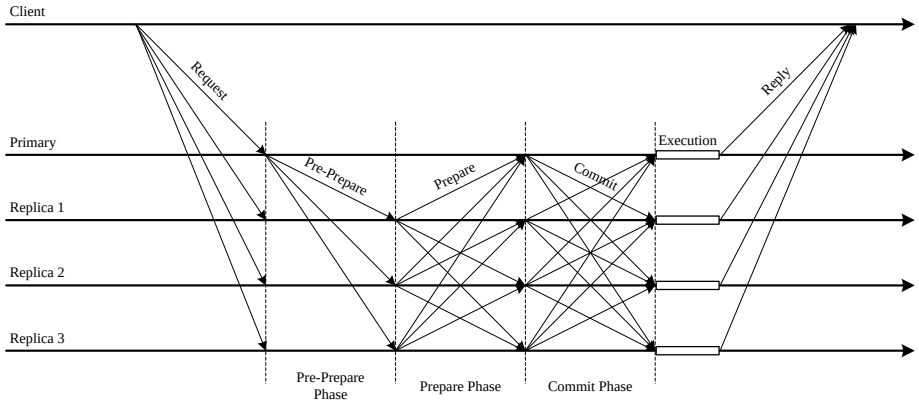


Figure 1 Normal Operation of the BFT Algorithm ($f = 1$)

If a replica cannot complete the three-phase algorithm by a predefined time or it receives an invalid message from the primary, it initiates a view change by sending out a view change request to all replicas to try to select a new primary in a round-robin fashion. If a correct replica receives $f + 1$ view change requests, it will join in even it is in normal operation state. The view change can be concluded when $2f + 1$ replicas agree on the view change requests and then a new view will be established with a new primary. View change is necessary to guarantee Byzantine agreement can eventually be reached among all correct replicas.

2.2 Strong Replica Consistency

The BFT algorithm described previously ensures the consistency of all correct replicas despite Byzantine faults only when the replicas behave deterministically. However, many practical distributed systems exhibit nondeterministic behaviors.

The antagonistic terms determinism and nondeterminism are from philosophy and have been extensively used in different areas. The following definition of determinism is from Wikipedia

*"**Determinism** is the philosophical position that for every event exist conditions that could cause no other event."*[30]

In computer science, an operation is said to be deterministic only when with the same input from an initial state, in absence of any failures, it always concludes to the same output. Deterministic operations are predictable. No matter how many times they are repeated, the results of a deterministic operation should always be consistent. In the domain of distributed systems, the term "replica determinism" means that when the same sequence of operations are applied at the replicas in the absence of failures, all server replicas should produce identical outcomes and move from the same initial state to the next consistent state. Replica determinism is a system wide property which is used to ensure all service replicas behave correctly even when they are running on completely different machines.

Replica nondeterminism, on the other hand, is not predictable. Even start from the same initial state in a failure free environment, server replicas still might perform differently with applying the same set of requests in exactly the same order. Nowadays,

nondeterminism plays more and more important roles in web applications or services. For example, many online gaming applications contain nondeterminism whose values proposed by one replica and cannot be verified by others (e.g., random numbers that determine the state of the applications). As another example, multi-threaded applications may exhibit nondeterminism (e.g., the thread interleaving) whose values cannot be determined prior to the execution of a request (without losing concurrency). All nondeterminism or replica nondeterminism should be carefully handled in web applications and services to ensure strongly consistence in the Byzantine fault tolerance system.

2.3 Software Transactional Memory

Software Transactional Memory (STM), as an alternative way to lock-based synchronization, is a concurrent control mechanism to protect the critical sections and guarantee transaction atomicity during concurrent processing in the multi-thread environments.

Lock-based synchronization mechanisms require the key to grant access to the protected section. A processing thread acquires the key before entering and releases the key after it finishes the operations in the protected section. If the key has been granted by a transaction, all later transactions will be blocked until the key has been released. Then the next transaction can request the key. In this way, the system is protected with mutual exclusions on critical sections. STM, on the other hand, allows concurrent access and

resolves the dependence problem dynamically. A thread can tentatively read or write the same shared memory regardless of what other threads might be doing and only makes the changes permanent after the validation is done during the commit phase. Otherwise, if conflicts have been detected, related transactions might be aborted and restarted until there is no more conflict.

There is no absolute sense of which solution is better between lock-based synchronization and STM since they thrive in different environments. STM enables optimistic concurrency, but adds the validation and retry overhead. This overhead highly depends on the number of shared objects the transaction has read, and grows linearly with the increases of the number of shared objects. With simpler and less error prone, STM is a better choice in the normal situation and it allows multiple threads to work on same pieces of data simultaneously. In the worst case, theoretically, the time complexity of STM is linear which will be the total time of all transactions plus the overhead associated with them.

2.4 Concurrent and Speculative Execution

In the last decade, the microprocessor technology has made tremendous advances. Now, it is very common that servers are equipped with multi-core Central Processing Units (CPUs) and sometimes with even multiple CPUs with great power of executing program instructions simultaneously. Parallel (concurrent) computing and concurrency

control are widely used in all areas in computer science to improve overall system performance.

Concurrent execution (concurrent computing) becomes a form of computing for which the programs are designed as collections of process units that can be executed simultaneously. Concurrent execution can be enabled on a single core CPU machine by interleaving the executions in a time-slicing or priority ordering way, but more ideally, it should be run on a multi-core machine with real parallelism at multiple cores by assigning the different process units to different computational cores. Parallel computing programs are much harder to design than the sequential programs. The challenges in concurrent execution design include not only making the processing more efficient, but also performing sound concurrency control, such as controlling the correct sequence of the interactions, accesses to shared resources. Concurrent executions are also hard to verify because they introduce several new obstacles that only exists in parallel computing such as race conditions, and these bugs are hard to detect since they only happens on special conditions. With concurrent execution and appropriately control, the overall system performance is optimized to a new level.

Speculative execution further extends the idea of optimization, where the system pre-executes the programs to utilize the CPU power more efficient. The pre-executed tasks might not even be actually needed. Speculation is using CPU idle time to pre-do work before the work is confirmed to be necessary, so as to prevent a delay for executing time after usage confirmation. The pre-execution takes risk. If it turns out that the work is not needed after all, the pre-execution will be wasted and totally ignored. The objective of speculative execution is to further improve the performance by utilizing extra

resources beyond the requirements. It is being widely used in optimistic concurrent computing. However, if the unnecessary speculation takes mandatory resources, it, will slow down the whole process and waste the time and resources.

CHAPTER III

RELATED WORK

3.1 Related Work of Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) [13, 14] by Castro and Liskov ensures both safety and liveness provided that less than one third of the replicas become faulty. Since this seminal work of BFT [13, 14] is published, a number of alternative BFT algorithms [1, 20, 32] have been proposed.

Query/Update (Q/U) [1] is a BFT protocol that requires the use of $5f + 1$ replicas to tolerate up to f faults, which is more than that is required for PBFT [13, 14]. Clients broadcast the cached histories and their requests to the server replicas and all the replicas optimistically execute the requests without inter-replica communication. With Q/U, the performance of the system can be significantly improved in the fault-free situation since the requests can be accomplished within a single round of communication

between the client and server replicas. On the other hand, if the client gets conflicting results, it will inform the replicas and drive them back to a consistent state. Then the request will be re-executed again. With more service replicas, Q/U can process the requests with fewer message exchanges during normal operations. However, it does not work well in the presence of concurrent update requests.

Hybrid Quorum (HQ) [20] combines both the quorum and agreement approaches. The same as PBFT, it only requires $3f + 1$ different replicas. In contention free cases, replicas choose the ordering in the first round and process requests in the second round based on the quorum from $2f + 1$ replicas. If any conflicts are detected, HQ relies on the Byzantine agreement to order and execute the requests. Compared with Q/U, HQ requires fewer replicas, but needs more rounds during normal execution. However, both Q/U and HQ cannot batch concurrent requests and have high latency when conflict happens as pointed out in [50].

Zyzzyva [32] is a speculative Byzantine Fault Tolerance protocol. Unlike other BFT algorithms, such as [13, 14, 16], it does not require replicas to reach the agreement before processing the requests. Zyzzyva protocol executes and responds to the clients immediately with the speculative results. If all replicas produce the same results, clients will conclude the requests and accept the results. Otherwise, the correct replicas might be temporarily inconsistent and reply with different answers. Nonetheless, all correct replicas, with the help of clients, will reach final agreement, and the replies will guarantee to be committed eventually. Although Zyzzyva significantly improves the performance during normal fault-free operations, it demands a more complicated recovery scheme.

Furthermore, in all three algorithms (Q/U, HQ, and Zyzzyva [1, 20, 32]), the replicas need the help from the clients to achieve agreements. We are concerned about this approach because if the client is faulty, it may endanger the integrity of the replicated service.

3.2 Related Work of BFT for Nondeterministic Applications

Replica nondeterminism has been studied extensively under the benign fault model [4, 6, 5, 39, 40, 41, 44, 46, 48, 56, 67]. However, there is no systematic classification of common types of replica nondeterminism, and even less so on the unified handling of such nondeterminism. [5, 46, 48] did provide a classification of some types of replica nondeterminism. However, they largely fall within the types of wrappable nondeterminism and verifiable pre-determinable nondeterminism, with the exception of nondeterminism caused by asynchronous interrupts, which we do not address in this dissertation.

The replica nondeterminism caused by multithreading has been studied separately from other types of nondeterminism, again, under the benign fault mode only, in [4, 6, 39, 40, 41, 44, 51]. These studies provided valuable insight on how to approach the problem of ensuring consistent replication of multithreaded applications.

It is realized that what matters in achieving replica consistency is to control the ordering of different threads on access of the same shared data. The mechanisms to record and to replay such ordering have been developed. So do those for checkpointing

and restoring the state of multithreaded applications (for example, [31]). Even though these mechanisms alone are not sufficient to achieve Byzantine fault tolerance for multithreaded applications, they can be adapted and used towards this goal. In this dissertation, we show when to record and (partially) verify the ordering, how to propagate the ordering, and how to provision for problems encountered when replaying the ordering, all using the Byzantine fault model.

As mentioned previously, the mechanisms developed in other research work regarding replica nondeterminism for Byzantine fault tolerance are limited to control a small subset of common replica nondeterminism, which we refer to as wrappable and verifiable pre-determinable replica nondeterminism [13, 14, 15, 16]. In BASE [16], it was recognized that a BFT system can be made more robust (to minimize deterministic software errors) by adopting a common abstract specification for the service to be replicated. A conformance wrapper for each distinct implementation is then developed to ensure that it behaves according to the common specification. Furthermore, an abstraction function and one of its inverses are needed to map between the concrete state of each implementation and the common abstract state.

In [14], Castro and Liskov provided a brief guideline on how to deal with the type of nondeterminism that requires collective determination of the nondeterministic values. The guideline is very important and useful, as we have followed in this dissertation research. However, the guideline is applicable to only a subset of the problems we have addressed. The problem of having to deal with non-verifiable nondeterminism is unique to the Byzantine fault model.

3.3 Related Work of Proactive Recovery

Ensuring Byzantine fault tolerance for long-running systems is an extremely challenging task. The pioneering work in the context of Byzantine fault tolerance is carried out by Castro and Liskov [13, 15, 52]. Our work is inspired by their work. However, the proactive recovery scheme in [13, 15] has a number of issues as we mentioned briefly in introduction.

First, it assumes that a simple reboot (i.e., power cycle of the computing node) can be the basis for repairing a compromised node, which might not be the case because some attacks might cause hardware damages, as pointed out in [52].

Second, even if a compromised node can be repaired by a reboot, it is often a prolonged process (typically over 30s for modern operating systems). During the rebooting step, the BFT service might not be available to its clients (e.g., if the rebooting node happens to be a non-faulty replica needed for the replicas to reach a Byzantine agreement).

Third, there lacks coordination among replicas to ensure that no more than a small portion of the replicas (i.e., no more than f replicas in a system of $3f + 1$ replicas to tolerate up to f faults) are undergoing proactive recovery at any given time, otherwise, the service may be unavailable for extended period of time. The static watchdog timeout used in [13, 15] also contributes to the problem because it cannot automatically adapt to various system loads, which means that the timeout value must be set to a conservative value based on the worst-case scenario. The staggered proactive recovery scheme in [13,

15] is not sufficient to prevent this problem from happening if the timeout value is set too short.

Recognizing these issues, a number of researchers have proposed various methods to enhance the original proactive recovery scheme.

The issue of uncoordinated proactive recovery due to system asynchrony has been studied by Sousa et al. [53, 54]. They resort to the use of a synchronous sub-system to ensure the timeliness of each round of proactive recovery. In particular, the proactive recovery period is determined a priori based on the worst case execution time so that even under heavy load, there will be no more than f replicas going through proactive recovery. The impact of proactive recovery schemes on the system availability has also been studied by Sousa et al. [55] and by Reiser and Kapitza [48].

In the former scheme, extra replicas are introduced to the system and they actively participate message ordering and execution so that the system is always available when some replicas are undergoing proactive recovery. However, the recovering replicas are regarded as failed, and therefore, higher degree of replication is needed to tolerate the same number of Byzantine faults and all the replicas would have to participate the Byzantine agreement process. In the latter scheme [48], a new replica is launched in a different virtual machine by the hypervisor on the same node when an existing replica is to be rebooted for proactive recovery so that the availability reduction is minimized. However, if an attack has caused physical damage on the node that hosts the replica to be recovered, or it has compromised the hypervisor of the node [58], the new replica launched in the same node in the scheme [48] is likely to malfunction.

Proactive recovery for intrusion tolerance has been studied in [2, 38] with the emphasis of confidentiality protection using proactive threshold cryptography [11]. Reboot is also used as the basis to recover compromised replicas, which suggests such schemes may also suffer from similar problems as those in [13, 14]. The idea of moving expensive operations off the critical execution path is a well-known system design strategy, and it has been exploited in other fault-tolerant systems, such as [37, 45, 48].

3.4 Related Work of Concurrent Speculative BFT

The current approach to enable concurrent execution in BFT systems is by exploiting application semantics. In PBFT [13, 14], it is noted that read-only requests can be delivered without the need of total ordering.

In [28], Kotla and Dahlin proposed to exploit application semantics for higher throughput by parallelizing the execution of independent requests. They outlined a method to track the dependency among the requests using application specific rules. In [19], Distler and Kapitza further extended Kotla and Dahlin's work by introducing a scheme to execute a request on only a selected subset of replicas. This scheme assumes that the state variables accessed by each request are known, and that the state object distribution and object access are uniform.

In prior work [17, 60], we proposed to rely on deeper application semantics to not only enable more requests (such as those that are commutative) to be executed

concurrently, but also minimize the number of Byzantine agreement steps used in an application (particularly for session-oriented applications).

This research takes a drastically different approach from those mentioned above. Rather than resorting to the application semantics, which may be expensive to acquire accurately and hard to reuse, we rely on the use of software transactional memory to dynamically capture the dependency of concurrent operations automatically. This approach is inspired by the work of Brito, Fetzer, and Felber [33], where a similar idea was used to ensure multithreaded execution for actively-replicated event stream processing systems. Our work applies the idea in a different context (i.e., Byzantine fault tolerance instead of crash fault tolerance) and furthermore, we carry out detailed experiments and analysis on the level of concurrency that can be achieved under various conditions.

CHAPTER IV

CONTROLLING REPLICA NONDETERMINISM FOR BFT

State-machine based Byzantine fault tolerant replication requires the replicas to operate deterministically, i.e., given the same request issued by a client, all replicas should produce the same reply provided that the replicas are in the same state prior to processing the request. However, all practical applications contain some degrees of nondeterminisms. When such applications are replicated to achieve fault tolerance, the nondeterministic operations must be controlled to reach strong replica consistency. Otherwise, an adversary may be exploiting the potential inconsistency to compromise the integrity of the replicated services.

In this chapter, we introduce our classification of common types of replica nondeterminism and present the system models and mechanisms for controlling these types of replica nondeterminism for distributed Byzantine fault tolerance (BFT) systems.

4.1 Classification of Replica Nondeterminism

To better understand and handle nondeterminism in distributed systems, we classify the replica nondeterminisms based on different properties.

First, we introduce a special type of nondeterminism termed as wrappable nondeterminism.

- *Wrappable nondeterminism*. A type of nondeterminism whose effects can be mapped into some pre-specified abstract operations and states which are deterministic.

Wrappable nondeterminism can be easily controlled by using an infrastructure-provided or application-provided wrapper function, without explicit runtime inter-replica coordination. For example, replica-specific identifiers, such as hostnames, process ids, and file descriptors, can be determined group-wise before the application is started. Another situation is when all replicas are implemented according to the same abstract specification, in which case, a wrapper function can be used to translate between the local state and the group-wise abstract state, as described in [16].

In this dissertation, we do not provide further discussion on the wrappable nondeterminism since it can be dealt with by a deterministic wrapper function without inter-replica coordination, and also because it has been thoroughly studied in [16].

Besides wrappable nondeterminism, we distinguish the rest of replica nondeterminisms based on two properties, determinable or verifiable.

Determinable is a property of nondeterminism based on the time when a particular operation will know the nondeterministic values. Using this as the criterion, we have pre-determinable nondeterminism and post-determinable nondeterminism:

- *Pre-determinable nondeterminism*. A type of replica nondeterminism whose values can be known prior to the execution of a request and it requires inter-replica coordination to ensure replica consistency.
- *Post-determinable nondeterminism*. A type of replica nondeterminism whose values can only be recorded after the request is submitted for execution and the nondeterministic values won't be complete until the end of the execution. It also requires inter-replica coordination to ensure replica consistency.

Nondeterminism verifiability is another criterion for classification. It is on whether a replica can verify the nondeterministic values proposed (or recorded) by another replica. According to this criterion, the nondeterminism can be divided into two different categories termed as verifiable nondeterminism and non-verifiable nondeterminism:

- *Verifiable nondeterminism*. A type of replica nondeterminism whose values can be verified by other replicas.
- *Non-verifiable nondeterminism*. A type of replica nondeterminism whose values cannot be completely verified by other replicas. Note that a replica might be able to partially verify some nondeterministic values proposed by another replica. This would help reduce the impact of a faulty replica.

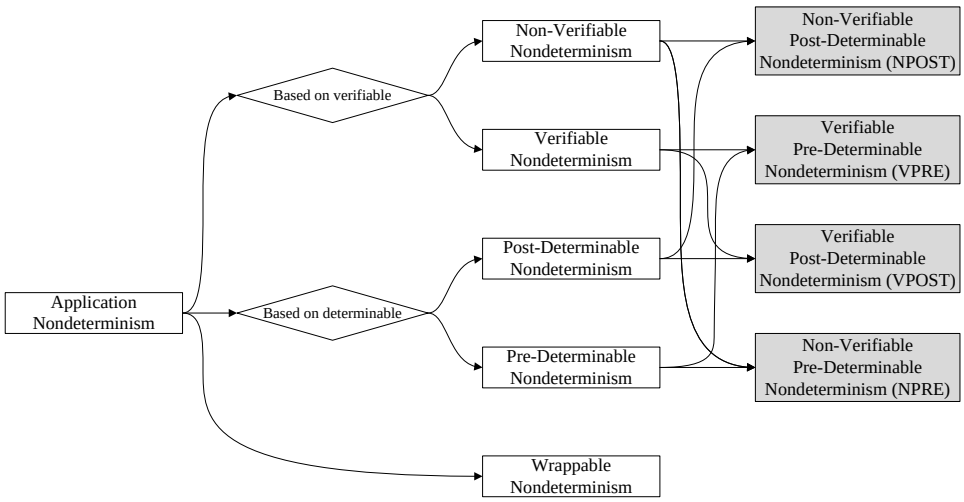


Figure 2 Classification of Nondeterminism

By using both criteria, as shown in Figure 2, we have four types of replica nondeterminisms of our interest:

- *Verifiable pre-determinable nondeterminism (VPRE)*. In the past, clock-related operations have been treated as this type of operations. However, strictly speaking, it is not possible for a replica to verify deterministically the proposal sent by another replica for the current clock value without imposing stronger restriction on the synchrony of the distributed system (e.g., bounds on message propagation, request execution, and the clock drifts).
- *Non-verifiable pre-determinable nondeterminism (NPRE)*. Online gaming applications, such as Blackjack and Texas Hold'em, exhibit this type of nondeterminism. The integrity of services provided by such applications depends on the use of good secure random number generators. For the best security, it is essential to make one's choice of a random number unpredictable, let alone verifiable by other replicas.

- *Verifiable post-determinable nondeterminism (VPOST)*. We have yet to identify a commonly used application that exhibits this type of nondeterminism. We include this type for completeness.
- *Non-verifiable post-determinable nondeterminism (NPOST)*. In general, all multithreaded applications exhibit this type of nondeterminism. For such applications, it is virtually impossible to determine which thread ordering should be used prior to the execution of a request without losing concurrency.

All four types of nondeterminisms have to be carefully controlled to guarantee system consistence. We will introduce our mechanisms to handle each of them later in this chapter.

4.2 System Model

The system model we considered is a client-server based application in an asynchronous network. Certain synchrony is necessary, similar to [13, 14], to achieve liveness which means the upper bound of the message transmission and processing delay has been asymptotic limited. We dynamically set this bound explored in the BFT algorithm as every time a view change occurs the timeout for the next view change is doubled.

Most frequently, both the client and the server, we believe, should be under normal operations. However, in very little cases, both of them could fall into Byzantine faults. We replicate the application server on $3f + 1$ different nodes to tolerant up to

f failures and each of replicas is modeled as a state machine. All servers are required to run or rendered to run deterministically even with some level of nondeterminisms, as we clarified before. To handle nondeterminisms, two critical issues to be resolved in the state-machine replicated system: the total ordering of requests and the required nondeterministic values. We are using BFT framework developed in [13, 14] to achieve the total ordering of the requests.

In the next section, we describe how we integrate our mechanisms into the BFT algorithm to control replica nondeterminisms so that all correct replicas will reach strong consistence on both the message ordering and the nondeterministic values.

4.3 Controlling Nondeterminism

Now, we introduce our mechanisms to handle common types of nondeterminisms.

4.3.1 Controlling VPRE Nondeterminism

If an operation contains Verifiable pre-determinable nondeterminism (VPRE), the primary replica proposes the nondeterministic values in the ndet parameter. Then both the nondeterminism type and obtained value are multicast in the PRE-PREPARE message to backups.

A replica verifies two critical parts when it receives a PRE-PREPARE message, the type and the values of nondeterminism. The nondeterminism type in clients' request

should be consistent with the one reported by the primary, and the nondeterministic values proposed by the primary is consistent with the replicas'. If both of validation process success, the backup replica accepts the PRE-PREPARE message from primary with ordering information and the nondeterministic values, and multicasts a PREPARE message to all other replicas. Otherwise, the replica suspects the primary and initializes a view change. From now on, the rest of the algorithm works the same as the original BFT algorithm, with the digest of the nondeterministic values included in both the PREPARE and the COMMIT messages. Figure 3 illustrats the details on how to control VPRE nondeterminism.

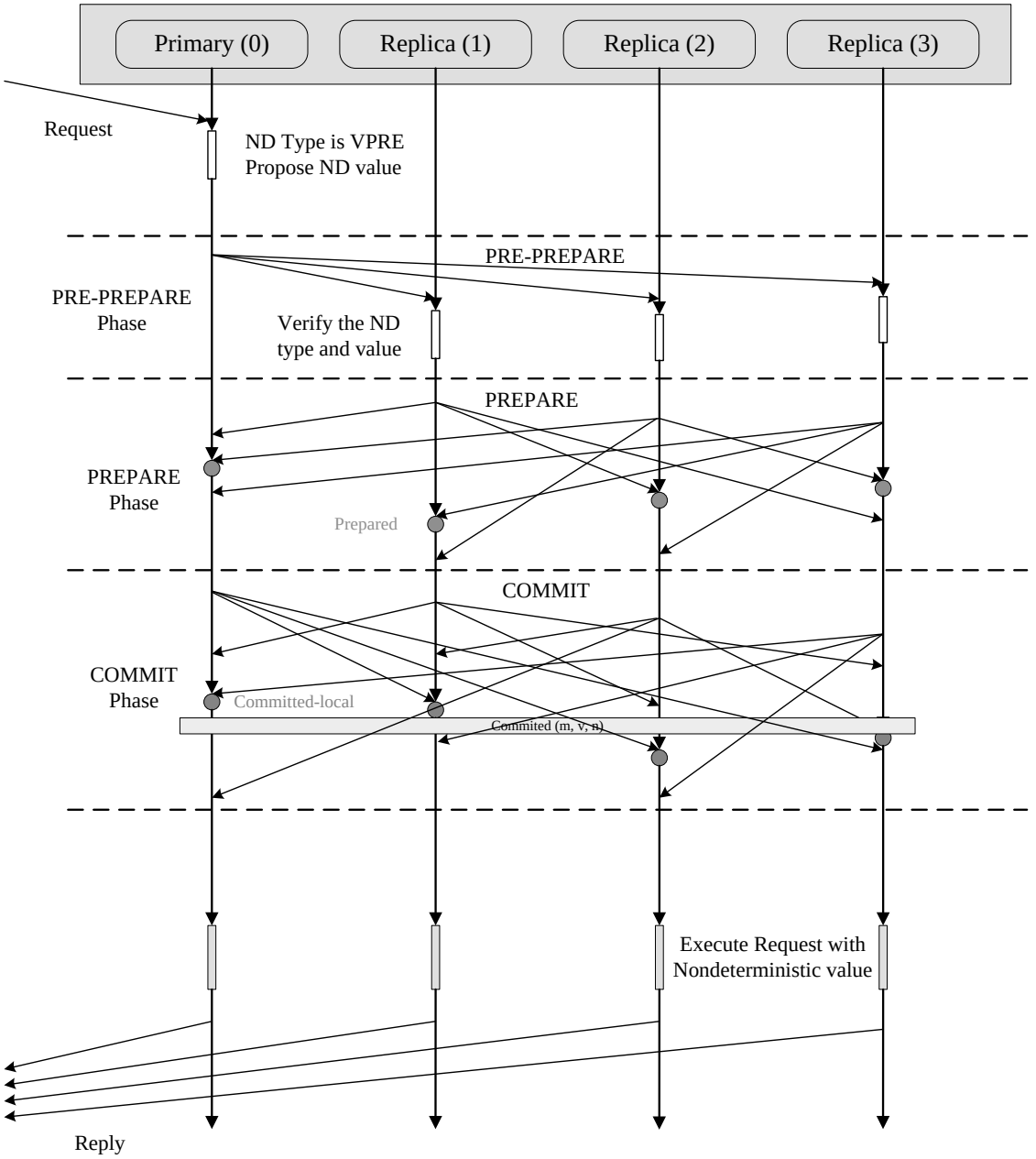


Figure 3 Normal Operation of the Modified BFT Algorithm for VPRE

4.3.2 Controlling NPRE Nondeterminism

If the nondeterminism for the operation at the primary is of type non-verifiable pre-determinable nondeterminism (NPRE), an extra phase, pre-prepare-update phase, is added to handle the nondeterminism. The idea behind it is to let every replica contributes its share of nondeterministic values which can prevent potential damage from any faulty replicas injecting the predicable value as nondeterminism.

When the primary gets the request with the type of NPRE nondeterminism, it proposes its share of nondeterministic values and multicasts the PRE-PREPARE message which includes both the type and the values of the nondeterminism to all backup replicas.

On receiving the PRE-PREPARE message, on top of original BFT, a backup replica only verifies the type of nondeterminism supplied by the primary is the same as the one included in the original request from clients. If the verification is successful, the backup replica builds a PRE-PREPARE-UPDATE message including its own share of NPRE nondeterministic values, and sends the PRE-PREPARE-UPDATE message back to the primary. Also the backup retrieves the nondeterministic values from primary and save it for later usage.

The primary expects $2f$ PRE-PREPARE-UPDATE messages from different replicas for a single request. These $2f$ PRE-PREPARE-UPDATE messages contain $2f$ different sets of contributions for NPRE nondeterministic values. Including the one from primary, $2f + 1$ set of contributions with proposer's digital signature protection will be sent from primary to backup replicas in a PRE-PREPARE-UPDATE message. When a replica gets the valid PRE-PREPARE-UPDATE message from the primary, it will

replace the old nondeterministic value with the new one it calculated based on all contributions. From now on, the BFT algorithm operates as the traditional way, except that both the PREPARE and COMMIT messages, the same as VPRE, also carry the digest of the nondeterministic values, and the $2f + 1$ sets of nondeterministic values are delivered to the application layer as parameters of the execution. The normal operation of the modified BFT algorithm for NPRE nondeterminism is illustrated in Figure 4.

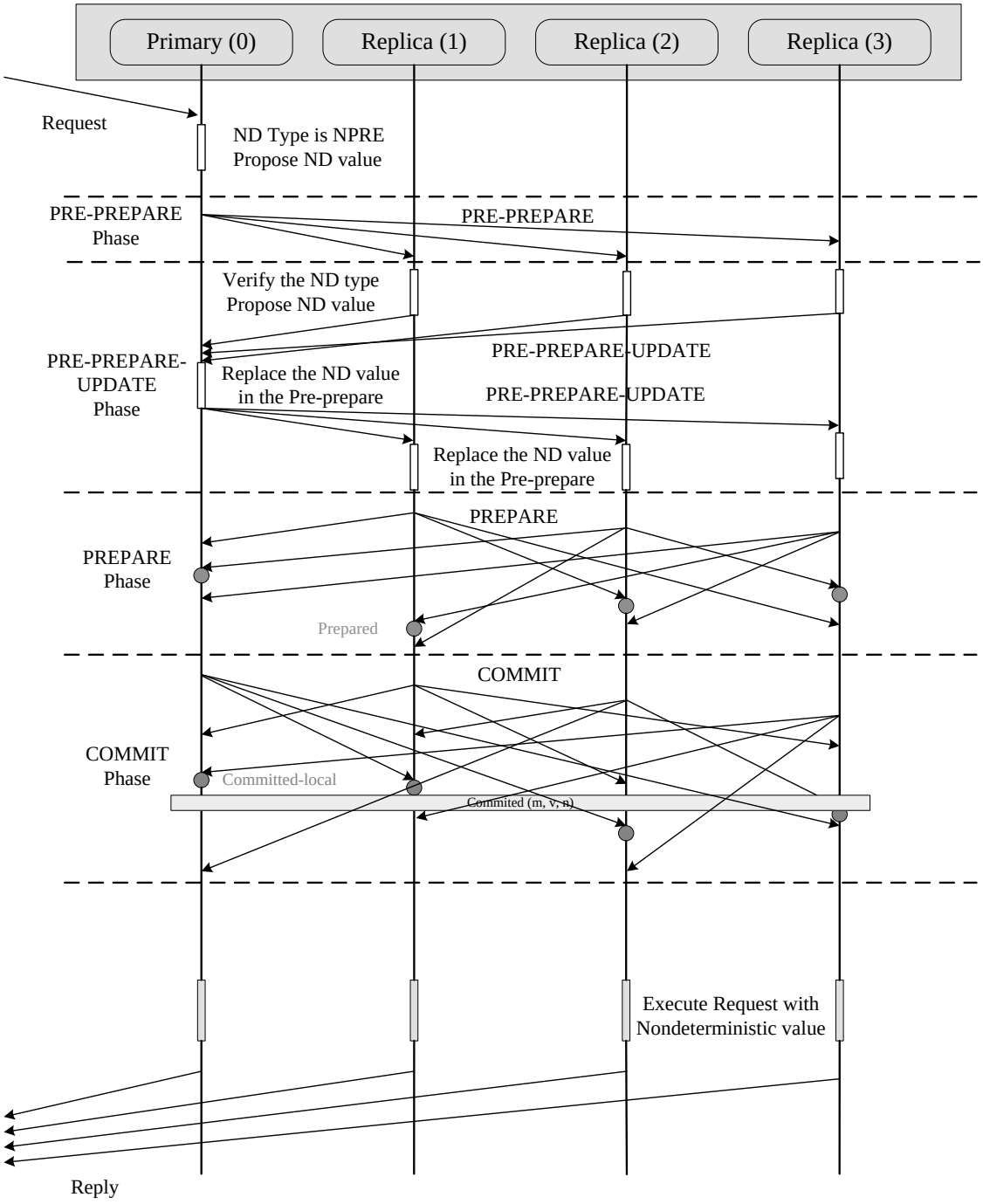


Figure 4 Normal Operation of the Modified BFT Algorithm for NPRE

4.3.3 Controlling VPOST Nondeterminism

In section 4.3.3 and section 4.3.4, we introduce the mechanisms to deal with POST-determinable nondeterminisms. To handle either verifiable post-determinable nondeterminism (VPOST) or non-Verifiable post-determinable nondeterminism (NPOST), the post-commit phase is necessary. Different from the pre-prepare-update phase for controlling NPRE, the post-commit phase involves the whole life-cycle of the Byzantine fault tolerance algorithm for correct replicas to reach an agreement on the nondeterministic values, which means that three rounds of control message exchanges are required similar to the way to determine the total ordering of requests under normal operation.

For VPOST, the primary, in the first round of Byzantine Agreement, includes only the nondeterminism type along with the ordering information in the PRE-PREPARE message without any nondeterministic values. The PRE-PREPARE message is multicast to all backup replicas to start the BFT algorithm. On receiving the PRE-PREPARE message, a backup replica checks the nondeterminism type after verification of the client's request and the ordering information. If the validation succeeds, the process will proceeds as usual to the prepare and the commit phases.

When an agreement is reached on the total ordering and nondeterminism type, the request message is delivered for execution at primary. As Post-determinable nondeterminism, a recorded nondeterministic value is expected as well as reply message. Once the primary returns from the execution, it sends the reply back to the client and builds a postnd log including nondeterministic values and the digest of the reply. This

postnd log will be send to backup replicas to verify whether the primary has actually generated the reply with the corresponding nondeterministic values. This starts the post-commit phase.

In the post-commit phase, the focus will be the nondeterministic values as well as the reply message, since we are not worrying about ordering information any more. With another round of BFT algorithm, all correct backup replicas should agree on the same set of nondeterministic values from primary and the values will be used in their own execution. Then the backup replicas produce a reply and compare the digest with the one from primary. If either second Byzantine Agreement cannot be reached or message digest mismatch, the primary will be suspected. However, the request will still be delivered for execution if the agreement reached on the nondeterministic values and the replica will send the reply back to client regardless of digest comparison result. This is because, replicas believe, the client will get the expected replies if all correct replicas execute the request with the same nondeterministic values, even different then the primary. Figure 5 shows the details about the normal operation of the modified BFT algorithm we used to handle VPOST.

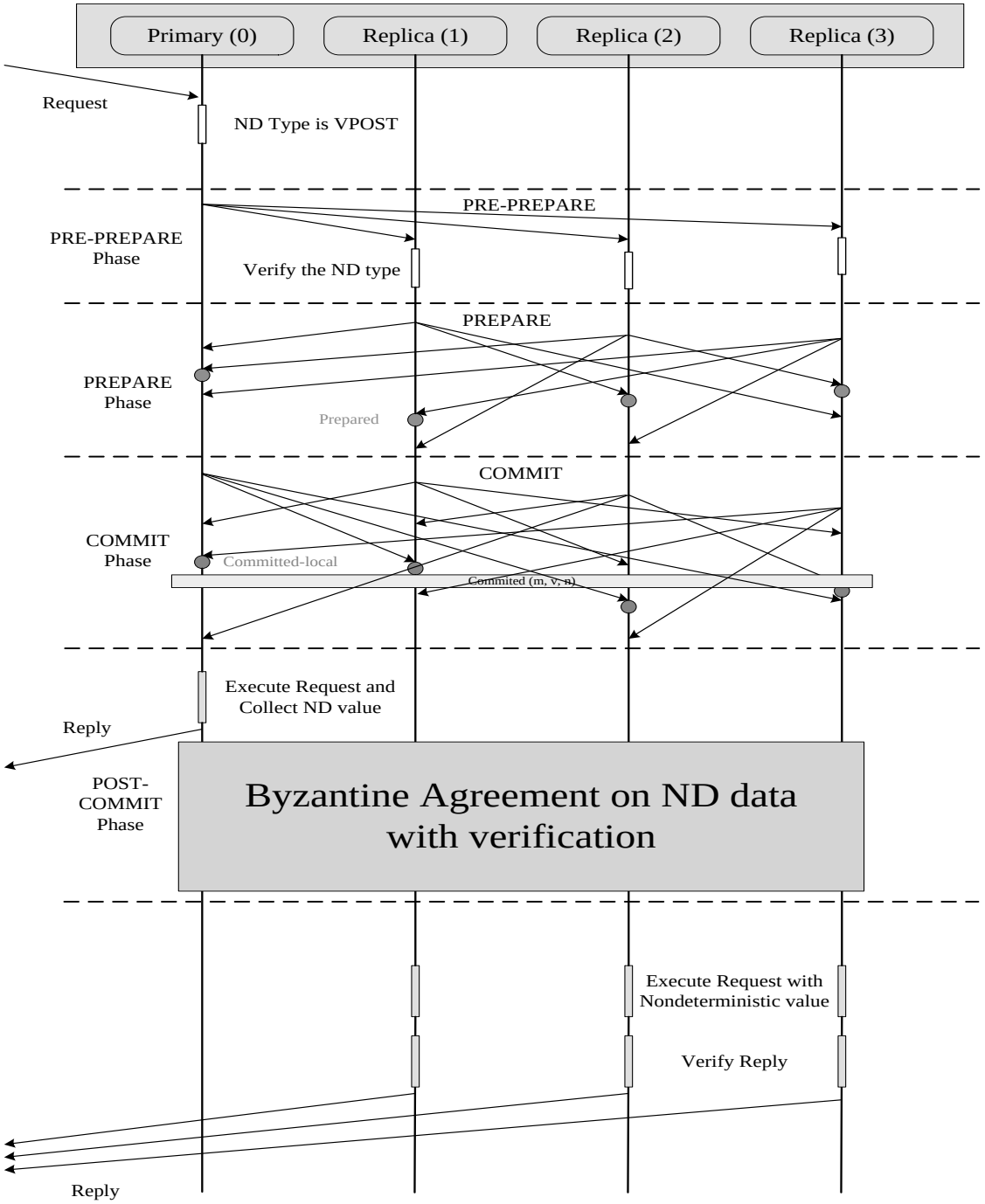


Figure 5 Normal Operation of the Modified BFT Algorithm for VPOST

4.3.4 Controlling NPOST Nondeterminism

To handle the non-verifiable post-determinable nondeterminism (NPOST), we use the same strategy as described in the previous subsection for controlling VPOST until a backup replica is ready to deliver the request to the application layer, as shown in Figure 6.

In contrast to VPOST, we have one more concern here. A faulty primary may disseminate some unexpected nondeterministic values to try to either confuse the backup replicas, or block them from providing useful services to the clients. For example, if the nondeterministic values are about thread interleaving, a faulty primary might provide the information in such a way to lead the backup replicas to deadlock or racing condition which might make the system crash. This is because the replicas, in general, cannot completely verify the correctness of the nondeterministic values until it actually executes the request. To prevent the system from crashing, we lunch a monitoring thread, as governance, separately with the main execution thread. And this monitoring thread can recover the replica when it runs into crash failures.

On the other hand, if the main thread can successfully complete the execution, then the backup replicas performs the same reply verification procedure as that described in the previous subsection.

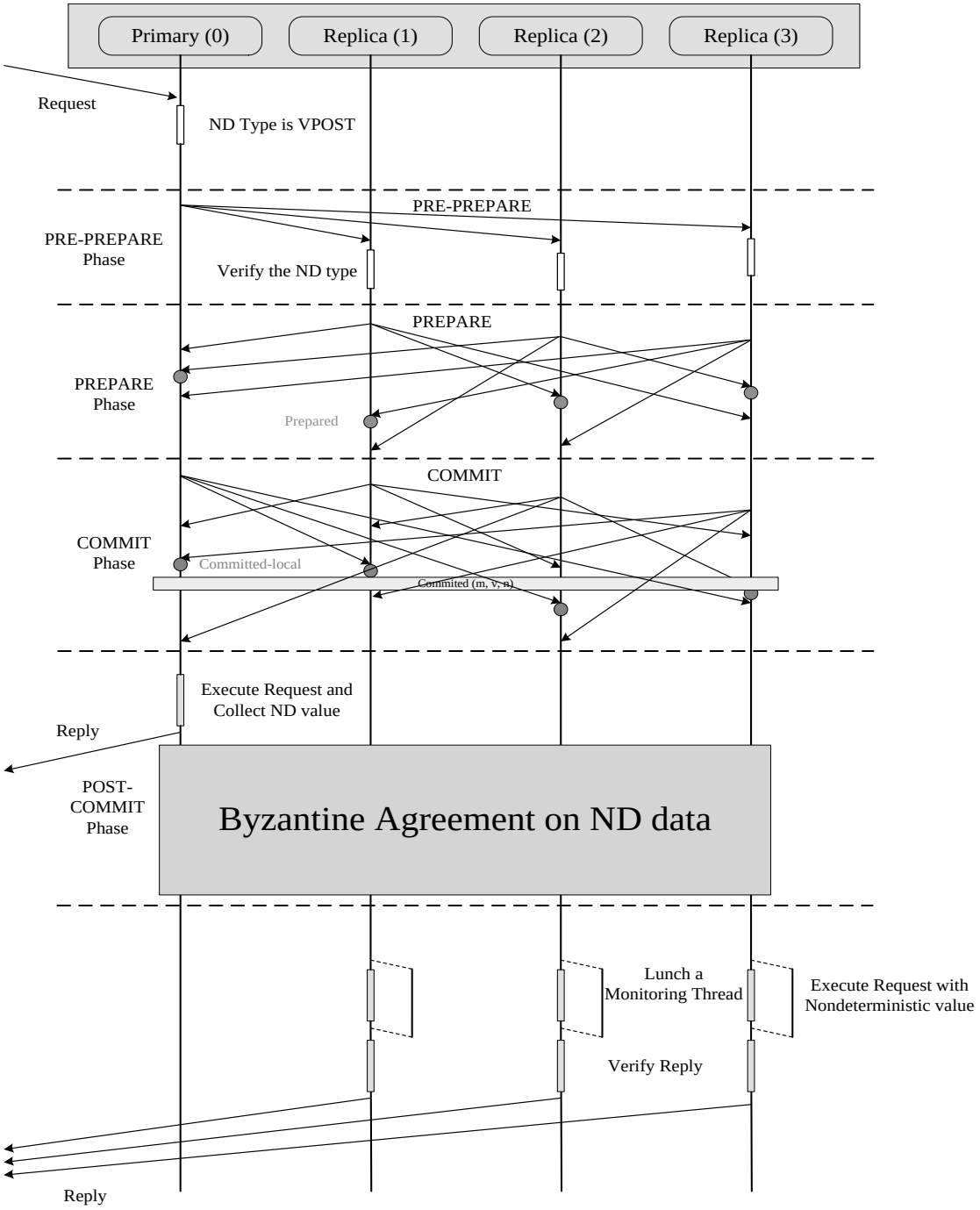


Figure 6 Normal Operation of the Modified BFT Algorithm for NPOST

4.4 View Change

A faulty primary might prevent a non-faulty replica from reaching a Byzantine agreement on either the ordering information and/or the associated nondeterministic values. When the backup replicas suspect the primary by any reason, they will start a view change to select a new primary. It might take more than one round of view changes for the whole process to select a new non-faulty primary and reach the Byzantine agreement in the new view. Moreover, during the view change, it is very important to carry over the adequate information from one view to another so that replicas could reach agreement on the same ordering information and nondeterministic values in different views.

The view change mechanism involves three control messages, consisting of VIEW-CHANGE, VIEW-CHANGE-ACK, and NEW-VIEW. A non-faulty replica initializes the view change if one of the following cases is true: (1) its view change timer expires; (2) it suspects either the ordering information or the nondeterministic values (for verifiable nondeterminism); (3) backups generate a different reply with primary (for post nondeterminism); (4) it receives $f + 1$ view change requests from other replicas. The basic view change flow we are using is the same as view change mechanism from original BFT algorithm. Besides we add the sets of information about states of post-pre-prepared and post-prepared in previous views (for Byzantine agreement on post-determinable nondeterminism).

To initialize view change, a replica updates all associated data in the log and then, based on records in its log, constructs a VIEW-CHANGE message. Upon multicasting

the VIEW-CHANGE messages, the replica cleans the log files since they are no longer useful. A replica accepts the VIEW-CHANGE message and replies a VIEW-CHANGE-ACK to the new primary of next view if the VIEW-CHANGE has all the information for current or an earlier view.

If the new primary in next view collects a VIEW-CHANGE message and $2f$ corresponding VIEW-CHANGE-ACK messages, it stores them as an entry with each entry is for a different replica. When the new primary has entries for $2f + 1$ replicas, it builds a NEW-VIEW message by using the data in the entries and broadcasts it to all other replicas. The NEW-VIEW message contains the start state of the new view as checkpoint, all requests with the sequence number start from the checkpoint to the most recent, and the associated nondeterministic values. All the information in NEW-VIEW is required to reach the agreement across different views. The new primary chooses the checkpoint from the information in the entries that the sequence number greater or equal to its own low water marker with support from at least $f + 1$ non-faulty replicas. If any requests, nondeterministic values, or checkpoints are missing from local, the new primary may fetch the state from other replicas. Start from the checkpoint, all requests later will once again go through the Byzantine agreement determination procedure as the way we introduced in section 4.3. For post-determinable nondeterminism, as an exception, if only ordering information is built in previous views without post-nondeterministic values, the NULL value will be included in NEW-VIEW message and the new primary will be responsible to propose new values to be used during the requests re-execution, which also requires the post commit phase to reach agreement.

When a backup replica, in the new view, receives a NEW-VIEW message, it validates the view change by comparing the proof in NEW-VIEW message with its own collection of VIEW-CHANGE messages. If a VIEW-CHANGE message missing locally, the replica requires a proof of correctness from new primary including the original VIEW-CHANGE message and $2f$ acknowledgements associated. Then, after validation is confirmed, the replica rebuilds a NEW-VIEW message with local information and compares it with the one received from new primary. If the verification passes, the normal operation resumes, otherwise, another view change is initialized immediately until successful. Figure 7 shows the details of view change.

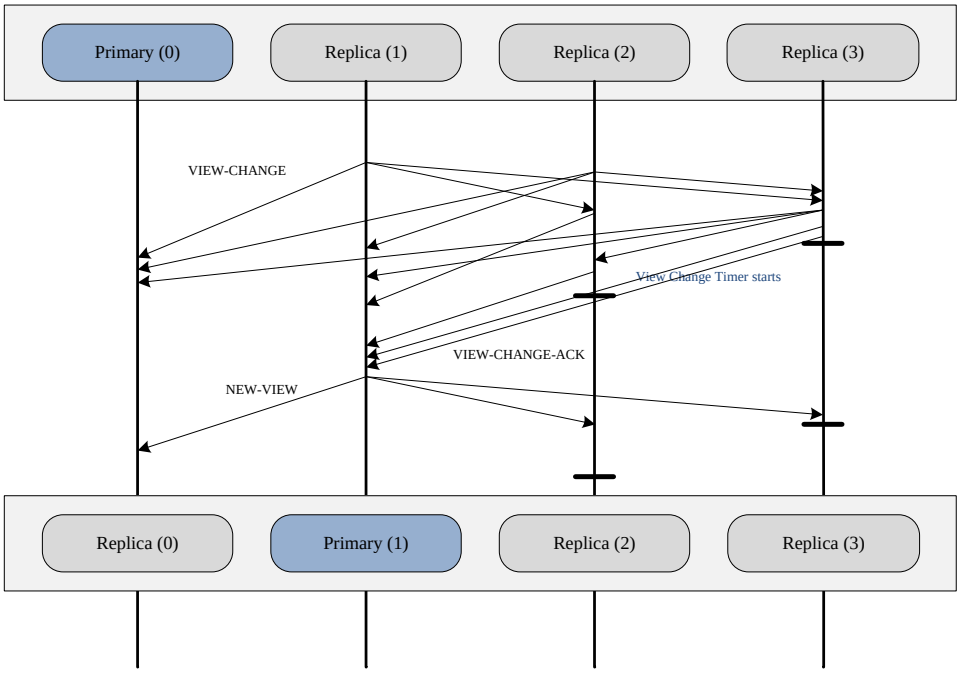


Figure 7 View Change

4.5 Implementation, Optimization, and Performance Evaluation

The implementation of mechanisms described in previous sections has been done in C++ and integrated into the BFT framework [13, 14, 15, 16]. A comprehensive experimental study has been carried out on the platform consists of 14 HP blade servers with each of them has two Quad-Core Intel Xeon 2GHz CPUs and 5GB memories. All blade servers are running Ubuntu Server 9 and are connected by Cisco Catalyst Blade 3020 Gigabit Ethernet Switch.

In performance evaluation, we focus on the overhead for providing controls on nondeterminisms in the BFT layer. The application layer work, such as cost associated with recording and verifying nondeterministic values, is not studied.

Furthermore, in practical applications, a request may involve more than one type of nondeterminism. Thus, we considered the possibility of composite types of nondeterminisms. Because we have yet identified any practical applications with VPOST, this type is omitted. The only types of nondeterminisms will be included in basic performance evaluations are listed as following.

- VPRE: Single type with verifiable pre-determinable nondeterminism
- NPRE: Single type with non-verifiable pre-determinable nondeterminism
- NPOST: Single type with non-verifiable post-determinable nondeterminism
- VNPRE: Composite type with both verifiable pre-determinable nondeterminism and non-verifiable pre-determinable nondeterminism
- VPRE-NPOST: Composite type with both verifiable pre-determinable nondeterminism and non-verifiable post determinable nondeterminism

- NPRE-NPOST: Composite type with both non-verifiable pre-determinable nondeterminism and non-verifiable post-determinable nondeterminism
- VNPRE-NPOST: Composite type with verifiable pre-determinable non-determinism, non-verifiable pre-determinable nondeterminism, and non-verifiable post-determinable nondeterminism.

In following sections, we first present, before performance evaluation, several optimizations we did to the mechanisms described previously. Then in basic performance evaluation, we use a single client with respect to all 7 types of nondeterminism listed above and various sizes of nondeterministic data (for clarity, we refer nondeterministic data as the set of nondeterministic values associated with each type of nondeterminism). Next, stress test, we present experiment results under various numbers of concurrent clients. In the final part, we report the impact of our mechanisms on the end-to-end latency during view changes.

4.5.1 Optimizations

All the results shown in following sections are obtained after optimization works, with which, the performance is significantly improved.

We optimized our mechanisms to handle NPRE nondeterminism. In pre-prepare-update phase, the primary will collect contributions of nondeterministic data from at least $2f + 1$ replicas and re-calculate a new one based on them. In PRE-PREPARE-UPDATE message from primary to replicas, the primary provides the proof of correctness including the collection of nondeterministic data used in re-calculation. Instead of multicasting the

whole nondeterministic data set, the primary disseminates the collection of digests, which will sharply reduce the message size especially when the data is large. Then the backup replicas could verify the digests from primary with its local copies. If a replica recognizes one or more missing proposed nondeterministic data locally, the retransmissions are required.

Another optimization we introduced is in the post-commit phase, which is used to handle NPOST nondeterminism. When we have multiple requests to process, we piggybacked the postn log, instead of a totally separate Byzantine agreement phase, with the PRE-PREPARE message of next request. In this way, we combine the Byzantine agreement for nondeterminism data of current request with the total ordering information of next request, which reduces the number of control message exchanges needed. Even though, the end-to-end latency for a particular request slightly increases, as a result, the overall throughput is significantly improved. If there is only a single request, as normal operation when the number of client is one, the post-commit phase still has to be done separately.

4.5.2 Basic Performance Evaluation

Error! Reference source not found. and Figure 9 show the summary of the end-to-end latency and throughput measurements for a client-server application under normal operation for different types of replica nondeterminism. For each iteration, a client issues a 1KB request to the server replicas and waits for the reply which will also be 1KB fixed size. When the client gets the valid reply, it sends out another request with no waiting

time between. For each run, we measure the total elapsed time for 100,000 consecutive iterations at client side, and calculate the average end-to-end latency and throughput.

The handling of different types of nondeterminism, except for VPRE, involves extra phases of message exchanges to reach agreements on both the ordering information and the nondeterministic data. As such, as shown in **Error! Reference source not found.** and Figure 9, the end-to-end latency is noticeably larger and the throughput is smaller, than that of VPRE. Furthermore, with larger size of nondeterministic data, the performance difference is more significant.

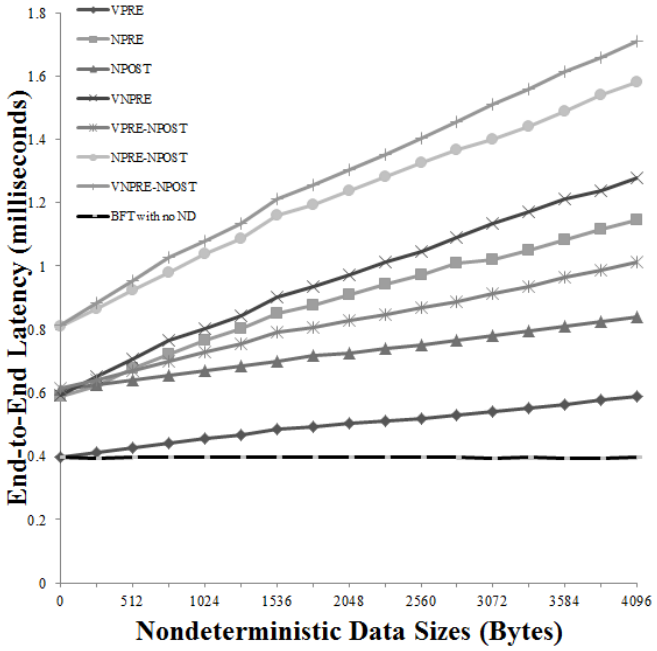


Figure 8 End-to-End Latency for Different Types of Nondeterminism under Normal Operations

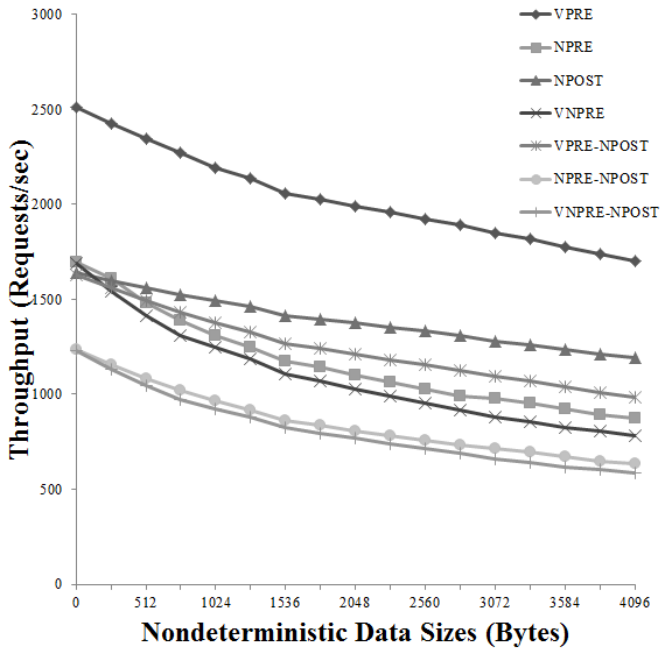


Figure 9: Average Throughput for Different Types of Nondeterminism under Normal Operations

With a closer look, one may notice a surprising scenario. There is a crossover for NPRE and NPOST in end-to-end latency. When the nondeterministic data size is small, the end-to-end latency of NPRE is smaller compared with NPOST. However, the latency for requests with NPRE grows rapidly when the nondeterministic data size increasing and becomes higher than that for the requests with NPOST eventually. This is because the pre-prepare-update phase, even with the optimization above, still involves at least two large messages while the post-commit phase has only one. For NPOST, it has a full Byzantine agreement loop including two more rounds of message exchanges than for NPRE, and this leads to a relatively large end-to-end latency when the nondeterministic data is small. However, following by increasing the size of nondeterminism data, the transmission delay for messages that contain large size of data takes dominate, it results

in much faster grow of end-to-end latency for NPRE, and eventually surpasses that for NPOST. The crossover for the throughput results shown in Figure 9 is due to the same reason.

4.5.3 Stress Tests

So far we did the experiments for normal operation with single client. In this section, as summarized in Figure 10 and Figure 11, we present the performance reports on stress tests with various numbers of concurrent requests from different clients. We use multiple clients issue the requests to the replicated server at the same time. Each of clients sends 100,000 requests consecutively, where the size of the nondeterministic data is kept at 256 Bytes.

With multiple concurrent requests, batching mechanism is enabled, which improves the overall throughput of the system. However, with larger number of clients, the waiting time increases. For a particular request, the latency becomes larger, as shown in Figure 10 and Figure 11.

Interestingly, there are other crossovers contained in the multi-clients performance diagram which also happens between NPRE and NPOST nondeterminisms. When the number of concurrent clients is less than 7, the type of NPRE is faster than NPOST. However, starting with 8 clients, the latency of NPOST becomes smaller. The reason for this phenomenon is because of the optimization we introduced previously. For NPOST nondeterminism, when there are sufficient number of concurrent clients, virtually all post-commit phase are combined with following requests. So it improves the

throughput for requests with NPOST nondeterminism when the number of clients increasing.

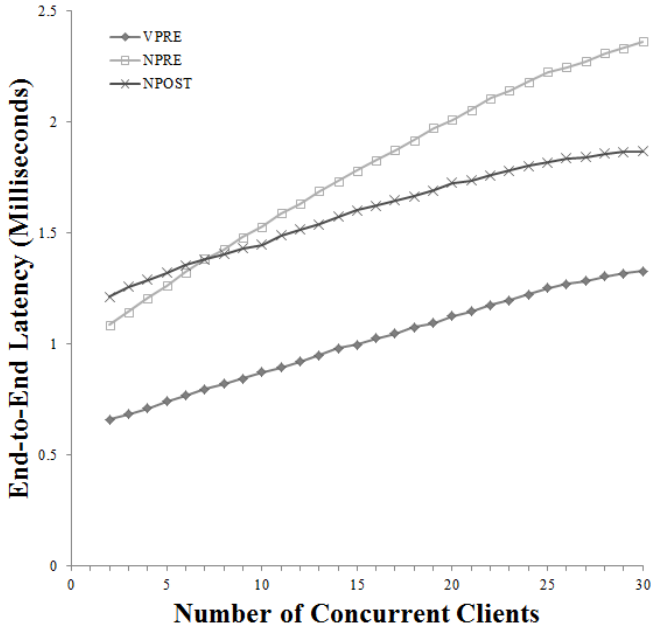


Figure 10 End-to-end Latency for Multiple Clients with Different Types of Replica Nondeterminism under Normal Operation

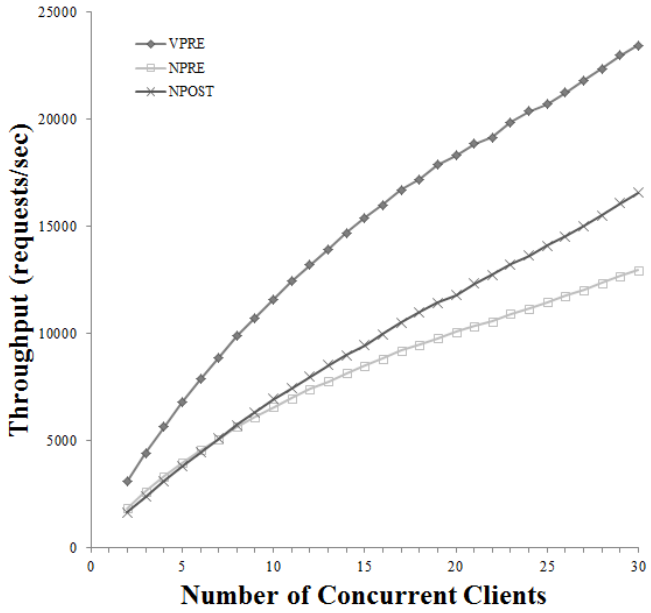


Figure 11 Throughput for Requests with Different Types of Replica Nondeterminism under Normal Operation

4.5.4 Impact on End-to-End Latency during View Changes

Until now, both basic performance and stress tests focus on the normal operations. In this section, we experience the impact of our mechanisms on the performance of view changes. In the experiment, a single client issues the requests to the replicated service. And some requests are instrumented so that they will crash the primary which will lead to a view change. We set the view change timer as 5-second, message retransmission timer as 150-millisecond and choose to use client side end-to-end latency, including the round-trip time of communication, the time used to detect the primary failure, and the view change latency, as the metric. Furthermore, a view change always succeeds and no message is lost during the view change.

Table 1 End-to-end Latency during View Changes

ND Data Size / ND Data Type	End-to-End Latency (seconds)					
	128 Bytes	256 Bytes	512 Bytes	1024 Bytes	2048 Bytes	4096 Bytes
BFT with no ND	5.303915					
VPRE	5.303713	5.303834	5.304212	5.304548	5.30449	5.304659
NPRE	5.304126	5.304294	5.304016	5.303665	5.30423	5.304159
NPOST	5.304225	5.304572	5.304388	5.304486	5.304382	5.304593

The view change experimental results are summarized in

Table 1. As can be seen, the end-to-end latency, for various scenarios, remains virtually equal, including the one without nondeterminism. This is what we expected since the mechanisms to handle different types of nondeterminism have very minimum impact on the view change. According to section 4.4, in modified view change mechanism, only the digest of nondeterministic data is piggybacked in the VIEW-CHANGE and NEW-VIEW messages. Furthermore, in our experiments, we assume there is no message lost during transactions. Therefore, from the performance point of view, it has virtually no negative effect.

4.6 Conclusion

In this chapter, we presented our mechanisms for handling common types of nondeterminism in a systematic and efficient manner based on the classification we introduced. The implementation of these mechanisms is carried out by extending the well-known BFT framework developed by Castro, Rodrigues, and Liskov [13, 16], which had very limited support for replica nondeterminism. Furthermore, we conducted

extensive experiments to evaluate the performance of our framework. And we show that our mechanisms only incur moderate runtime overhead.

CHAPTER V

PROACTIVE RECOVERY

State-machine based Byzantine fault tolerance (BFT) algorithms, including those designed to control replica nondeterminism described in the previous chapter, assume the availability of $3f + 1$ replicas to tolerate up to f faulty replicas. However, over the lifetime of a system, the number of faulty replicas may eventually exceed f in the presence of persistent adversaries. To ensure the reliability and the availability over extended period of time (typically 24x7 and all year long), proactive recovery [48, 53, 54, 55], where replicas are periodically restarted and repaired before they are detected to be faulty, becomes essential.

In this chapter, we present our proactive recovery scheme for BFT. Compared with the proactive recovery scheme proposed by Castro and Liskov [13, 15], the primary benefit of our scheme is a reduced vulnerability window under normal operation. This is achieved by two means. First, the time-consuming reboot step is removed from the critical path of proactive recovery. Second, the response time and the service migration latency are continuously profiled and an optimal service migration interval is dynamically

determined during runtime based on the observed system load and the user-specified availability requirement.

5.1 System Model

Our proactive recovery scheme partially relies on the synchrony of the system, i.e., the round of proactive recovery should be completed within a bounded time. However, all Byzantine agreement needed in the proactive recovery is guaranteed to be safe without any synchrony assumption.

Our service migration-based proactive recovery scheme includes three main components:

1. A pool of nodes for active server replicas. To tolerate up to f Byzantine faulty replicas, $3f + 1$ service replicas are needed in the active pool and they do all the operations as we discussed in the previous chapter.
2. A pool of standby nodes. The size of standby pool should be large enough ($> f$) to repair damaged nodes while enabling frequent service migration for proactive recovery.
3. A trusted configuration manager (similar to what has been described in [52]). This trusted configuration manager is to manage the pool of standby nodes, and to assist service migration. i.e., it is frequently probing and monitoring the health of each standby node, and repairing any faulty nodes detected.

Three main components are separated to different subnets which are connected by an advanced managed switch (i.e., Cisco Catalyst 6500) for faults isolation. Each node, either in active pool or standby pool, has three network interfaces NIC1, NIC2, and NIC3. We use NIC1 for connection to external network, NIC2 to connect between active and standby pools, and NIC3 for connection to the configuration manager. Although each node installs three network interfaces, only the ones in active pool have all three enabled. In standby pool, we disable NIC1 to make the nodes only accessible internally. Trusted configuration manager can dynamically control NIC1 and NIC2 of any node through NIC3, e.g., it can disable NIC1 on a node to remove it from active pool and switch NIC2 of the same node to add it in standby pool.

All server replicas may be subject to malicious faults in both active and standby pools. However, the majority attacks, we assume, are imposed from external networks. So the successful attacks on the standby nodes, which are isolated from external environment, should be much less likely than those on the nodes in the active pool. Similar to [54, 55], we assume only fail-stop model failures are on the trusted configuration manager and, to ensure high availability, the trusted configuration manager is replicated using the Paxos algorithm [35]. Other assumptions regarding the system still hold as we mentioned in the previous chapter.

5.2 Proactive Service Migration Mechanisms

The proactive service migration mechanisms ensure long term reliability and availability. The detailed objectives including:

- Ensure a consistent membership view for available standby nodes on each active replica;
- Determine the time and the method to start a migration;
- Select the source and the target nodes for migration;
- Transfer the correct state to the new active replicas;
- Notify the clients with the new membership after each proactive recovery.

5.2.1 Standby Nodes Registration

The nodes in the standby pool are controlled by the trusted configuration manager. Probing and sanitization procedures are applied on standby nodes periodically to ensure that they are not compromised. To ensure all the correct active replicas have the consistent membership of the available standby nodes, a refreshed standby node needs to notify all active replicas when the sanitization procedure are finished successfully. Otherwise, if the trusted configuration manager cannot repair the faulty nodes, a system administrator will be called to manually fix the problem.

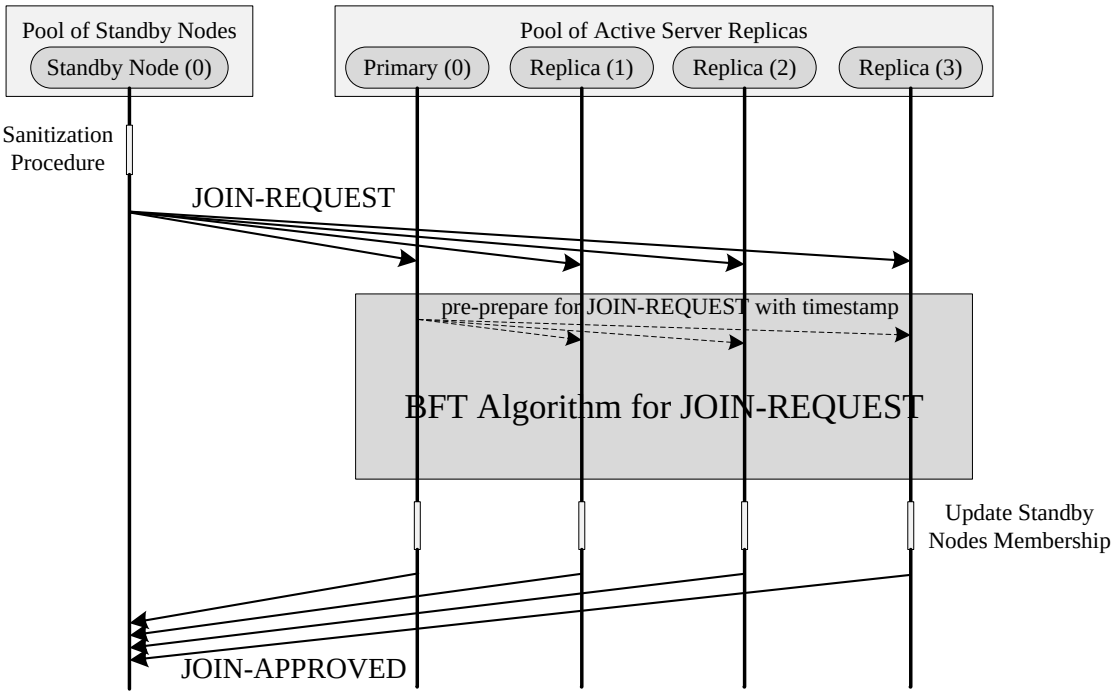


Figure 12 Standby Nodes Registration Protocol

The registration protocol is illustrated in Figure 12. A node in standby pool multicasts the JOIN-REQUEST, including a counter maintained by the secure coprocessor, to all active service replicas. An active replica will accept the JOIN-REQUEST if the request is the one with highest counter from the same standby node. When the primary gets the valid JOIN-REQUEST, it will assign a timestamp to the request as the join time and initialize a Byzantine agreement process. This process is important, so that all active nodes have the consistent membership view of the standby nodes. The significance of the join time will be elaborated later. When the JOIN-REQUEST has been committed, the correct active replicas will update its own standby nodes membership and send the JOIN-APPROVED reply back. The registration process completes if the requesting standby node gets $2f + 1$ matched JOIN-APPROVED messages from different active replicas.

A standby node might have gone through multiple rounds of proactive sanitization before it is selected to enter active pool and run an active replica. Every time the repairing procedure completes, a new registration is triggered to reconfirm the membership. The active replicas subsequently update the join time of the standby node if the registration process completes successfully.

Although standby nodes are much less likely to be compromised, it is still possible. When the configuration manager deems a registered standby node as faulty, an on-demand service repair will be initialized and the standby node is deregistered from the active replicas by sending a LEAVE-REQUEST. The LEAVE-REQUEST is handled in a similar way as that for JOIN-REQUEST.

5.2.2 Proactive Recovery

Proactive recovery will be triggered if either one of the two scenarios becomes true:

- The software-based recovery timer expires, or
- An on-demand service recovery is invoked by the trusted configuration manager.

A proactive recovery timer is started at the beginning of the service and is reset at the end of each round of migration. One of the advantages of our proactive service migration is that the recovery timer can be adjusted dynamically based on the synchrony of the system and the workload on the system. This benefit can prevent harmful excessive concurrent proactive recoveries and a potentially large window of vulnerability.

The migration timer is initialized when we start the replicated service and it requires several user specified parameters, including:

1. The target system availability A^0— the most important parameter.
2. The minimum number of requests p^0 served during a single round of proactive service migration.

Furthermore, to elaborate our algorithm on how to adjust the proactive recovery timer, we define some symbols here:

1. The response time to order and execute a request T_{oe}. Please note, T_{oe} does not include the queuing delay for the request being ordered.
2. The latency T_s to carry out a service migration, i.e., the time it takes to swap out an active replica and replace it with a clean standby replica.

The timeout value is initialized to $p^0 T_{oe}$, and during runtime, we continuously measure the average response time T_{oe} for the most recent p^0 requests and the service migration latency T_s. A notification is sent to the system administrator if either the response time or the service migration latency exceeds the worst-case values.

Based on the availability, we can calculate the service migration timeout value by the following equation:

$$T_\omega = \frac{A^0 T_s}{1 - A^0} \tag{4.1}$$

The parameter p^0 is defined by user. The migration timeout value T_ω is dynamically adjusted to $p^0 T_{oe}$, if $p^0 T_{oe} > \frac{A^0 T_s}{1-A^0}$. So to satisfy both the requirements on

the minimum number of requests served in each migration period and the target system availability, the migration timeout T_ω is set as following:

$$T_\omega = \max\left\{p^0 T_{oe},\ \frac{A^0 T_s}{1 - A^0}\right\} \tag{4.2}$$

On expiration of the migration timer, a replica chooses a set of f active replicas, and a set of f standby nodes to initialize the proactive recovery. f active replicas are selected based on the reverse order of their identifiers. For example, since we have $3f + 1$ active replicas, so in the fourth round, we have only one left which is required to be recovered next round. Then we select other $f - 1$ active replicas with identifiers from $3f$ to $2f + 2$. So replicas with id $0, 3f, \ldots, 2f + 2$ are selected. The set of f standby nodes is selected on the timestamp of the registration, the younger the better. We choose the ones with the latest timestamp because of the least probability of these nodes to have been compromised at the time of migration (assuming brute-force attacks by adversaries).

After making the decisions on the service migration sets, the replica multicasts an INIT-MIGRATION request to all others. There is a migration number contained in the initial migration request, which is determined by the number of successful migration rounds recorded by the replica. A correct replica accepts an INIT-MIGRATION message if all three conditions are hold:

(1) The INIT-MIGRATION message carries a valid authenticator;

(2) The receiver has not accepted another INIT-MIGRATION message from the same replica in the same view with the equal or higher migration number;

(3) Selected set of active replicas and set of standby replicas are consistent with the sets determined by the receiver according to the same migration set selection algorithm.

When a replica gets $2f$ consistent INIT-MIGRATION from different replicas, it will construct the MIGRATION-REQUEST message. The most important requirement of the proactive recovery is to ensure a consistent up-to-date state for service migration, which can be done by Sync-Point Determination Phase. In the Sync-Point Determination Phase, the migration requests are totally ordered with respect to normal requests, and the one with top priority will be processed immediately without queuing. The primary orders the MIGRATION-REQUEST in the same way as that for a normal request, except that

(1) It does not batch the MIGRATION-REQUEST message with normal requests, and

(2) It piggybacks the MIGRATION-REQUEST and $2f + 1$ INIT-MIGRATION messages, as proof of validity, with the PRE-PREPARE message. The reason for ordering the MIGRATION-REQUEST is to ensure a consistent synchronization point for migration at all replicas.

An illustration of the migration initiation protocol is shown in Figure 13.

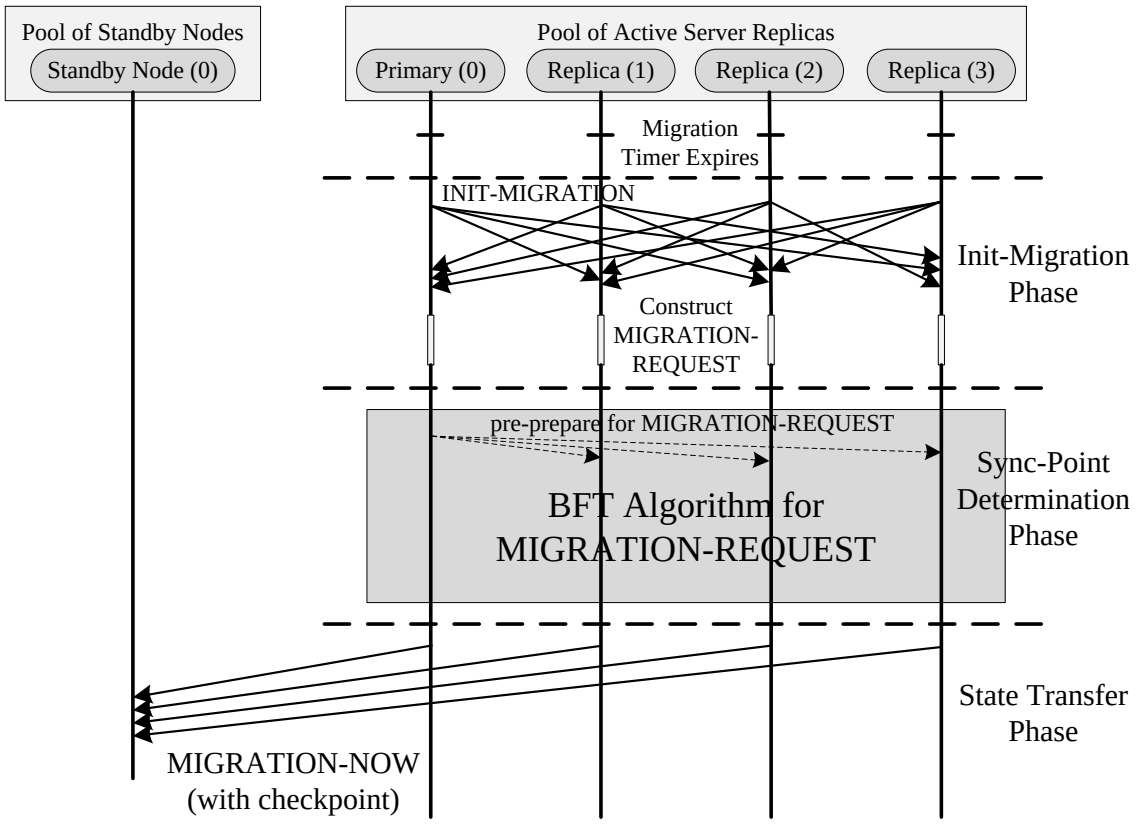


Figure 13 Proactive Service Migration Protocol

Each replica starts a view change timer after the MIGRATION-REQUEST message has been constructed. If it cannot receive the PRE-PREPARE message from current primary before the timer expiration, a view change will be initiated. The new primary should resume the proactive service migration.

Again, the MIGRATION-REQUEST message is totally ordered to ensure that all correct active replicas reach the same synchronization point when performing the service migration. The only difference with the normal requests ordering is that the replica must have all $2f + 1$ INIT-MIGRATION messages the primary used to construct the

MIGRATION-REQUEST, and verify the active node set and standby node set match those in the INIT-MIGRATION messages.

When a correct replicas reach the synchronization point, it takes the checkpoint of its state, including both the application and the BFT middleware state, and multicasts a MIGRATE-NOW message to the f standby nodes to be migrated and all replicas of the configuration manager. The MIGRATE-NOW message contains a set of tuples to identify the pair of source-node and target-node. The standby node that is designated as the target node will replace the active node indicated as the source node, once it completes the proactive recovery procedure.

A replica sends the actual checkpoint, together with all queued request messages if it is the primary, to the target nodes in separate messages. If a replica is to be recovered, its NIC1 interface is expected to be disabled and it stops accepting any new requests. However, this holds for correct replicas only. If the replica is faulty, it might not do so. This is the reason why the trusted configuration manager must be informed of the migration by all correct active replicas. When there are $f + 1$ MIGRATE-NOW notifications, the configuration manager changes the switch configuration to forcefully disable the NIC1 interface from the switch end and performs other sanitizing operations on the faulty nodes.

When a standby node collects $2f + 1$ matched MIGRATE-NOW requests, it is promoted to run as an active replica and then applies the checkpoint to its state. From now on, this node starts to participate in the normal operation and becomes a new valid active replica.

5.2.3 New Membership Notification

A faulty or potential faulty replica can be recovered by our alternative proactive recovery scheme. However, there is a lag between when a faulty replica has been migrated and when it has been sanitized by the configuration manager. In the meantime, the faulty replica still can send messages to the active replicas and the clients. Hence it is very important to inform the clients with the new membership of the active replicas so that they will ignore the messages received from the current active pool during the transition period.

To improve the performance, the NEW-MEMBERSHIP notification is performed in a lazy manner after the first request of the service migration has been processed. However, if the primary replica has been selected to be replaced, the notification should be sent immediately so the clients could send their requests to the new primary instead of the old one. Furthermore, the notification is sent only from original active replicas, not the new ones, because the clients do not know them yet.

5.2.4 On-Demand Migration

On-demand migration is invoked when one or more faulty nodes have been detected by either the trusted configuration manager or by a replica with the solid evidence. The mechanism is very similar for both the timer based service migration and the on-demand service migration, except the trigger itself and the faulty nodes selection, since for on-demand migration, the nodes to be sanitized are already decided. The

migration process is the same and both should start with the INIT-MIGRATION message.

5.3 Performance Evaluation

We implemented the proactive service migration mechanisms described in this chapter and integrated into the Byzantine fault tolerance (BFT) framework developed by Castro, Rodrigues and Liskov [13, 15, 16]. All the related operations are simulated in software. And furthermore, we didn't fully implement the trusted configuration manager since we lack the sophisticated hardware equipment to facilitate the subnet dynamic control.

The testbed of the experiments consists of a set of Dell SC440 servers with a Pentium dual-core 2.8GHz CPU and 1GB RAM. They are running SuSE Linux 10.2. Similar to [13], those are general-purpose servers without hardware coprocessors. All the components including the configuration manager, the three pools of replicas, and the clients are located in the same physical local area network connected with a 100 Mbps switch.

The motivation of the experiments is to evaluate the runtime performance of the proactive service migration scheme. The micro-benchmarking example included in the original BFT framework is adapted as the test application. Both the request and the reply messages are set to 1KB fixed length, and each client generates requests consecutively

without any think time. We are using a 1ms processing delay by busy loop to simulate some actual workload before it echoes back the payload to the client.

Due to the potential large state, we employed the following optimization: only one node sends the full checkpoint to the target node and the others send the digest of the checkpoint instead. The target node can verify the checkpoint by comparing the digest generated from the full copy with the ones received from other replicas.

We present two sets of experiments. First, the runtime cost of the service migration mechanism with a fixed migration timer. Second, characteristics on dynamically adjusted migration period with various conditions.

5.3.1 Runtime Cost of Service Migration

We present the runtime cost of the service migration schema by measuring the recovery time on a single node with various service state sizes. In each run, the service migration interval is kept at 10s. The recovery time is determined by measuring the time elapsed between the following two events:

(1) The primary sending the PRE-PREPARE message for the MIGRATION-REQUEST, and

(2) The primary receiving a notification from the target standby node indicating that it has collected and applied the latest stable checkpoint.

We refer to this time interval as the service migration latency. Figure 14 summarizes the service migration latency with respect to various state sizes and the number of concurrent clients. It is not surprising to see that the cost of migration is

limited by available bandwidth (100Mbps) since the time to take a local checkpoint and restore one is negligible in our experiments (memory operation). This is intentional for two reasons:

(1) The check point taking and restoration cost is very application dependent, and

(2) Such cost is the same regardless of the proactive recovery schemes used.

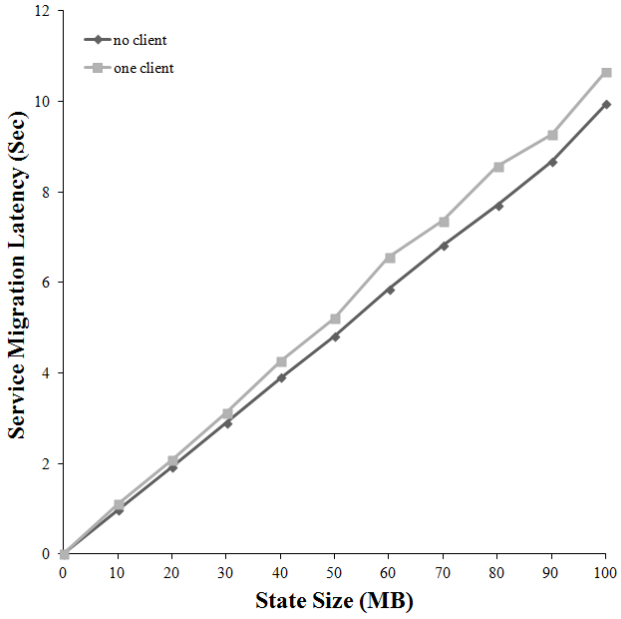


Figure 14 Service Migration Latency for Different State Sizes

Furthermore, we measure the migration latency as a function of the system load in terms of the number of concurrent clients. As can be seen in Figure 15, the migration latency increases more significantly for larger state when the system load is higher. When there are eight concurrent clients, the migration latency for a state size of 50MB is close to 10s. This observation suggests that if a fixed watchdog timer is used, the watchdog timeout must be set to a very conservative worst-case value. If the watchdog timeout is

too short for the system to go through four rounds of proactive recovery (of f replicas at a time), there will be more than f replicas going through proactive recoveries concurrently, which will decrease the system availability, even without any fault.

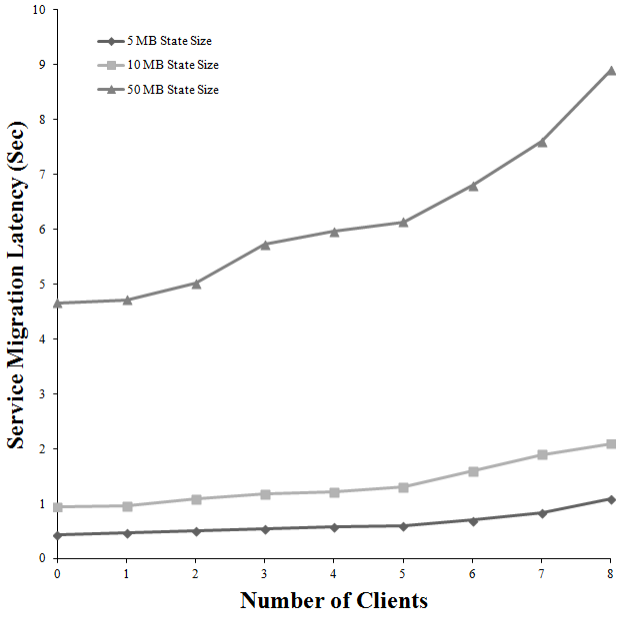


Figure 15 Service Migration Latency with Respect to the System Load

5.3.2 Dynamic Adjustment of Migration Interval

The objective of this set of experiments is to demonstrate the capability of dynamic migration interval adjustment. The results present how the migration interval changes under different system loads due to various concurrent clients and state sizes.

We provide following parameters as user specified:

1. Target system availability $A^0 = 83.3\%$

2. Minimum number of requests served during a round of proactive service migration $p^0 = 10{,}000$
3. Initial value of the service migration interval $T_\omega^0 = 50s$

Figure 16 shows the results of the dynamic adaptation of migration interval in the presence of a single client. As expected, when the state size is relatively small, 20MB or below, $p^0 T_{oe}$ is used because the migration latency is small and the user specified minimum requests needed to meet. As the state size increases, larger migration latency is needed to meet the availability requirement. Again, we show that the migration interval dynamically determined are much smaller than the worst-case value except when the state size is very large.

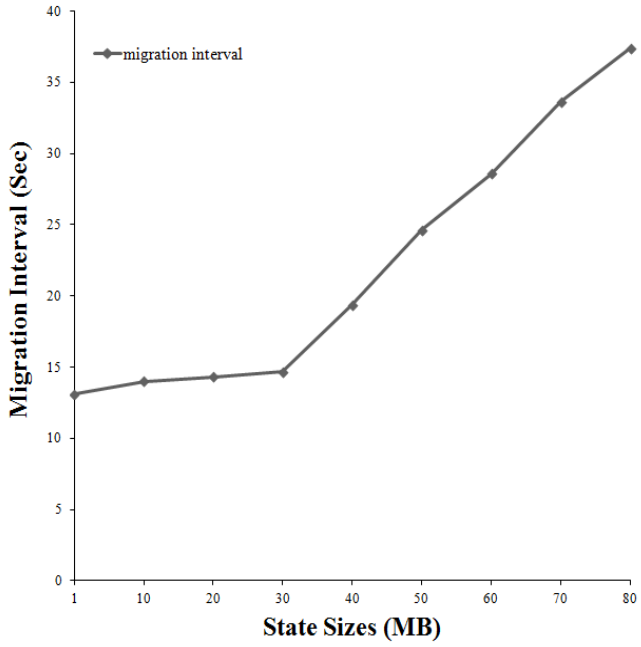


Figure 16 Dynamic Adaption of Migration Interval for Different State Size

Figure 17 shows us the migration interval with various numbers of concurrent clients. It may be surprising to note that the migration timeout value actually decreases when the number of concurrent clients increases for state sizes of 5MB and 10MB. This might appear to be counterintuitive. However, it can be easily explained. This is an artifact caused by the aggressive batching mechanism in the BFT framework [13] we used. With batching, the cost of ordering a single request is reduced. Consequently, the response time per request is reduced, which results in a smaller migration timeout value. (Recall that T_{oe} does not include the queuing delay of the request being ordered.)

Another interesting observation is that the migration timeout values determined at runtime are much smaller than the worst-case value except when the state size is large and the number of concurrent clients is significant. For many applications, their state size might gradually increase over time as they process more application requests. A larger state would mean larger migration latency, as indicated in equation 4.2.

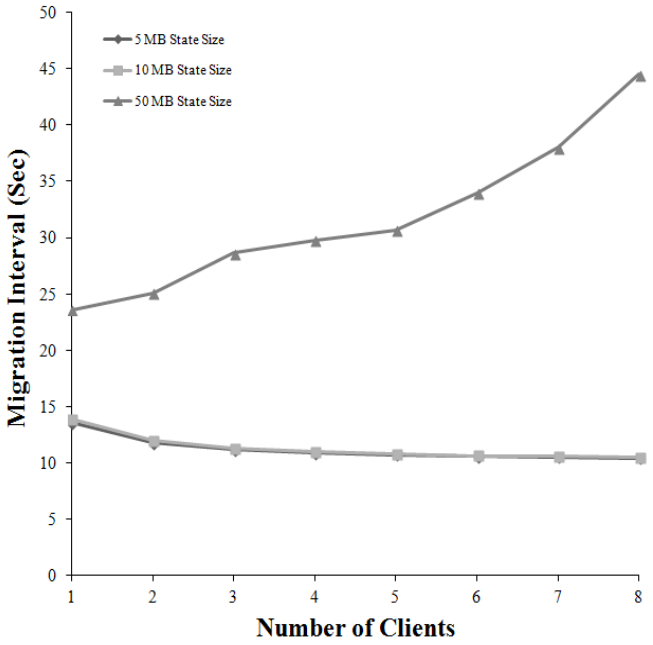


Figure 17 Corresponding Migration Interval with Respect to the Number of Concurrent Clients

5.4 Conclusion

In this chapter, we introduced a novel proactive recovery scheme based on service migration for long-running Byzantine fault tolerant systems. We described in detail the challenges and mechanisms needed for our migration-based proactive recovery to work. The primary benefit of our migration-based recovery scheme is a smaller vulnerability window during normal operation. When the system load is light, the migration interval can be dynamically adapted to a smaller value from the initial conservative value, which is usually set based on the worst-case scenario, and hence, resulting in a smaller vulnerability window. Our scheme also shifts the time consuming repairing step out of

the critical execution path, which also contributes to a less chance to compromise and a smaller vulnerability window. We demonstrated the benefits of our scheme experimentally with a working prototype.

CHAPTER VI

CONCURRENT BFT

In existing Byzantine Fault Tolerance (BFT) algorithms, application requests are executed one after another according to the established total ordering to ensure strong replica consistency. This inevitably limits the performance of the system without fully exploiting the multi-core processors that are pervasively available today. To lift the limitation, we incorporate the Software Transaction Memory (STM) technique into BFT systems. By using STM, it is possible to delivery multiple requests for concurrent execution as long as the commit order is controlled such that the order conforms to the total ordering of the requests that triggered the transactions, which is referred to as the ordering rule in this chapter. Furthermore, we can use the multi-version and commit barrier approaches to enable speculation to further improve the performance by pre-executing the requests and hold the result temporarily until the execution is validated. If, by any chance, the speculation is wrong, the system will rollback and re-execute the requests based on the correct order.

In this chapter, we introduce our concurrent and speculative BFT algorithm that could bring the performance of BFT system to a new level.

6.1 Conflicts Model

Conflicts management is a very important task during the concurrent execution of multiple transactions. To better understand it, we introduce the conflicts model in web applications first with examples. We do not discuss basic read/write or write/write conflicts in our conflicts model because they can be easily controlled. We only focus on conflicting operations that can pass the validation test, but may lead to the violation of the ordering rule.

For example, in a simple client-server application, each client sends a request to start a transaction and wait for a reply. If concurrent execution is not enabled, the transactions will be created and executed one after another sequentially, and requests may have to wait for their terns in a waiting queue. If the server has the capabilities of concurrent executing, multiple transactions, requested by different clients, can be triggered and processed at the same time. Figure 18 illustrates the basic idea.

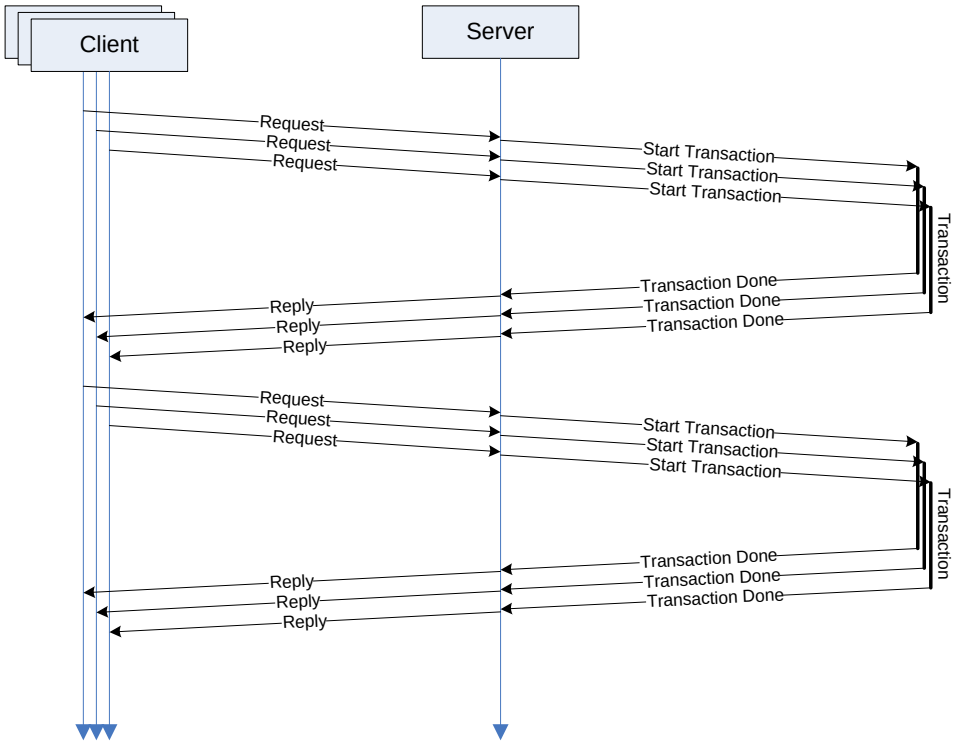


Figure 18 Concurrent Execution in a Client-Server Application

Concurrent execution is a double-edged sword. On the one hand, it might speed up the whole system performance in optimal conditions. On the other hand, concurrent transactions may have to be aborted when conflicts arise. Conflicts may be unavoidable when concurrent transaction processing is enabled. Some conflicts may be difficult to discover. In a stateful web application, concurrent transactions must be made equivalent to a sequential execution, and some transactions may have to be aborted when conflicts are detected. In the following, we elaborate several common types of conflicts.

The first type of conflicts: A transaction with higher timestamp concludes before another transaction with lower timestamp, and both transactions update the same piece of data successfully but in the wrong order. As shown in Figure 19, transaction T_i starts before transaction T_{i+1}, (i.e., $TS_{T_{i+1}} > TS_{T_i}$, where TS is the timestamp of the

transaction). They both update a shared data item. It may happen that T_i updates the shared data item after T_{i+1} has already committed cause by unexpected delay. In this case, although the execution of the two transactions is linearizable, the commit order of the two transactions violates the ordering rule because the transaction that has smaller timestamp is committed later than the one that has bigger timestamp. If uncontrolled, this conflict may cause replica inconsistency because it may happen that some replicas commit T_i ahead of T_{i+1} while some other replicas commit T_{i+1} ahead of T_i.

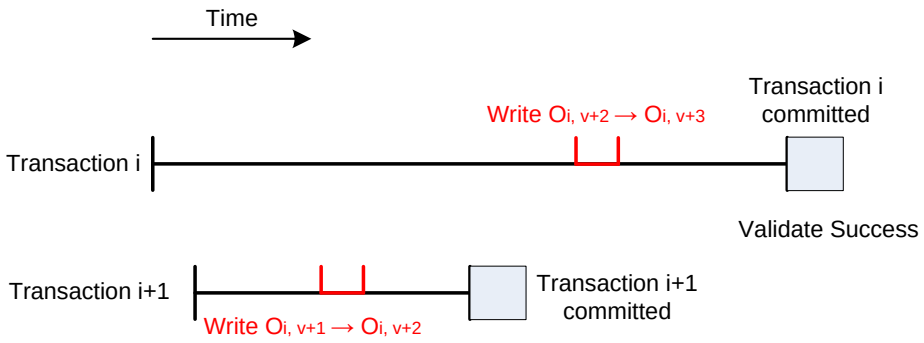


Figure 19 Conflict Model 1 – Update Shared Data in the Wrong Order

The second type of conflicts: STM uses commit timestamp to verify the transaction, and the one with lower timestamp is forced to re-execute due to the read/write conflicts. In the following example, we consider two concurrent transactions, transaction T_i and transaction T_{i+1} with $TS_{T_i} < TS_{T_{i+1}}$. T_{i+1} accesses the shared data item earlier than T_i. When T_{i+1} finishes its work and tries to commit, it detects the write/write conflicts with T_i and this would force T_i be re-executed. Again, although the two transactions are executed according to some lienarizable order, the actual order violates our ordering rule because it may cause replica inconsistency.

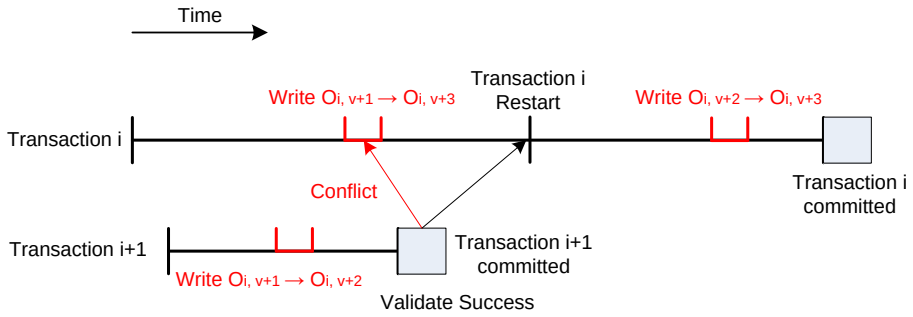


Figure 20 Conflict Model 2 – Early Transaction Forced to Abort and Restart

Note that for normal read and write conflicts, the conflict resolution rule defined by STM to ensure linearizable execution of concurrent transactions is adequate. Figure 21 below shows an example.

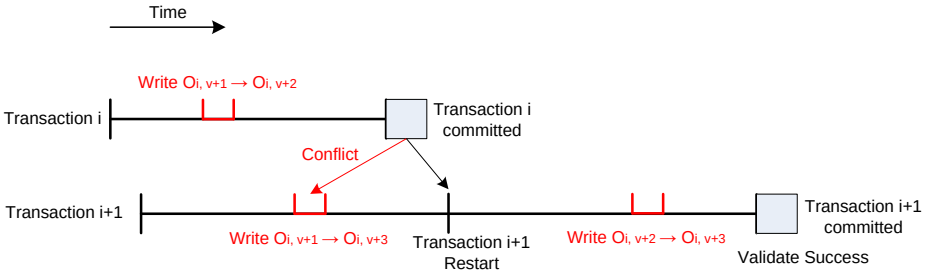


Figure 21 Normal Read and Write Conflicts Example

6.2 Speculative Concurrent BFT

The execution of concurrent transactions makes conflict unavoidable and we have introduced different types of conflicts in the previous section. The basic read/write and write/write conflicts have already been discovered and handled dynamically by STM.

And the most difficult part left for us is to add additional mechanism in the validation step to detect the violation of our ordering rule.

In this section, we introduce speculative concurrent BFT based on two strategies, namely, commit barrier, and multi-version speculation with commit barrier.

6.2.1 Commit Barrier

We impose the following rule to detect the conflicts in concurrent stateful systems:

"*When a conflict is detected, the transaction with the smaller timestamp or sequence number should be committed and, if necessary, the one with the larger timestamp or sequence number must be aborted and restarted*."

This rule must be abide by no matter how complicated the situation is. The reason why some transactions are valid for STM but violate our ordering rule is because STM doesn't track a specific relative ordering among the transactions. The transactions with higher sequence number could be committed earlier according to the STM conflict resolution rule. To prevent this from happening, we introduce a commit barrier, an extra stage of validation during the commit phase. With the commit barrier, the transaction can commit only when all transactions with lower sequence numbers have already been committed, otherwise, it has to wait.

We will reuse the two examples introduced in the previous section to see how to use the commit barrier to solve the problem. The two conflict models are handled in the same way during the commit barrier. The transaction T_{i+1} with higher timestamp or

sequence number reaches the commit point first, however, it cannot commit since we have a barrier now and transaction T_i has not committed yet, as shown in Figure 22. In this scenario, T_{i+1} has to wait. On the other hand, transaction T_i continues processing and, if the validation is successful, it can commit. After T_i has fully completed, the commit barrier releases T_{i+1}. However, when T_{i+1} is validated, a conflict will be detected. This would force transaction T_{i+1} to abort and to restart. In second try, T_{i+1} would be able to commit.

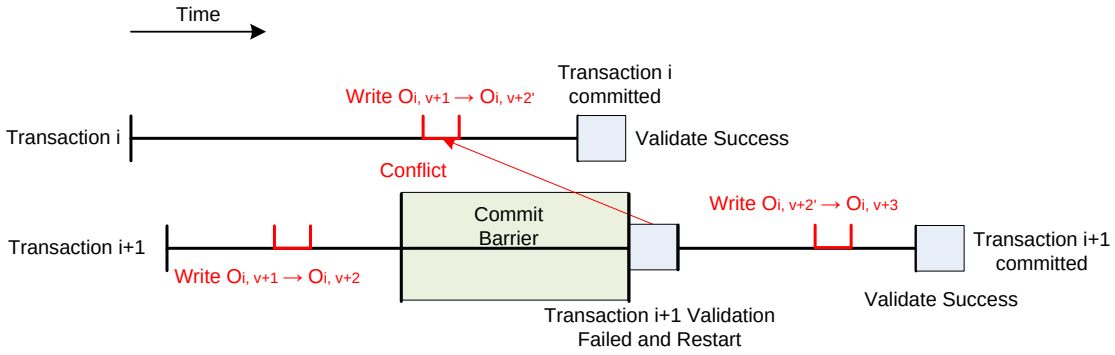


Figure 22 The Commit Barrier Solution for our Conflict Models

6.2.2 Multi-Version Speculation with Commit Barrier

Combining STM with commit barrier, we can guarantee the basic rule cannot be violated. violated. However, the use of the commit barrier may negatively impact the performance.

As the example shows in

Figure 23, when transaction T_{i+1} touches the shared data the very first time, we know there will be a conflict. With commit barrier, the problem can be solved but in efficiently. Hence, we propose another approach – Multi-version.

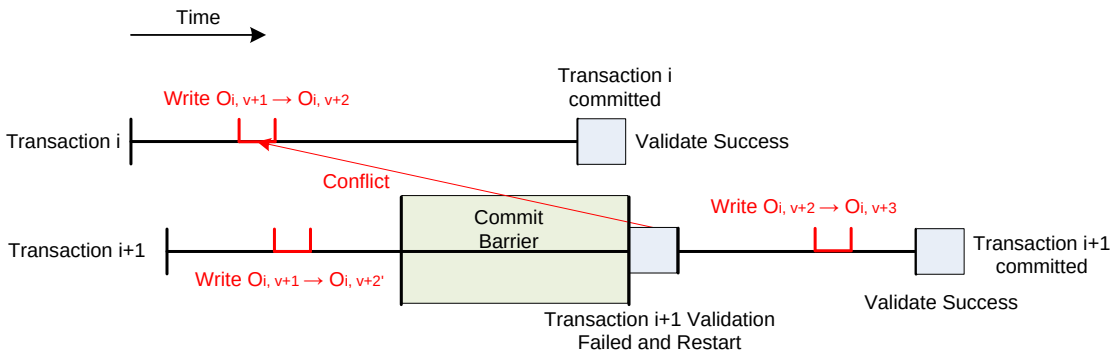


Figure 23 Commit Barrier Performance Issue

Multi-version has been used in different places. The basic idea is that we keep multiple versions for shared data instead of a single static one. The multiple versions will include the last committed and the tentative versions. The last committed version is the permanent data that has been committed successfully by a transaction. The tentative version, on the other hand, is the version produced when a data item is updated by another active transaction that has yet to be committed. Every time the transaction requires a shared data item, it fetches the most recent version of the data item, even it is the tentative version. Then the fetched latest version is used in the following operations.

During the validation step, the transaction has to confirm that the tentative data has already been committed. If that is not the case, the transaction has to be aborted and restarted.

Multi-version breaks the isolation property and exposes uncommitted data to all other active transactions. In our system, we store tentative data associated with the corresponding sequence number assigned to the transaction. When a transaction accesses the shared data, it prefers to use the one with the highest sequence number that is lower than that of the current transaction. If the transaction that produced the tentative version has been aborted, all transactions that are using the tentative version would also have to be aborted. Note that the multi-version approach must be used in conjunction with commit barrier validation to guarantee that the tentative version, if it is used, is committed before the transactions that accessed the tentative version.

Figure 24 illustrates how to apply the multi-version mechanism to an example scenario. Transaction T_i executes normally and it updates the shared data first. When transaction T_{i+1} accesses the same piece of data, there are two versions and transaction T_{i+1} fetches the tentative version from T_i, even though it is not permanent. The tentative version of data will be used in operations of T_{i+1}. Since T_{i+1} takes less time to finish, it reaches the commit point before transaction T_i but is blocked by the commit barrier. In the mean time, transaction T_i continues its processing and commits after validation which releases the commit barrier of T_{i+1}. Now instead of normal validation, T_{i+1} also needs to verify the data version it used to finally commit. After T_i has committed successfully, the tentative version becomes permanent. So T_{i+1} can also be committed directly without restart. By using multi-version, transaction T_{i+1} can commit right after the T_i and the

ordering rule will hold. Any conflict would be handled nicely with almost no negative performance impact.

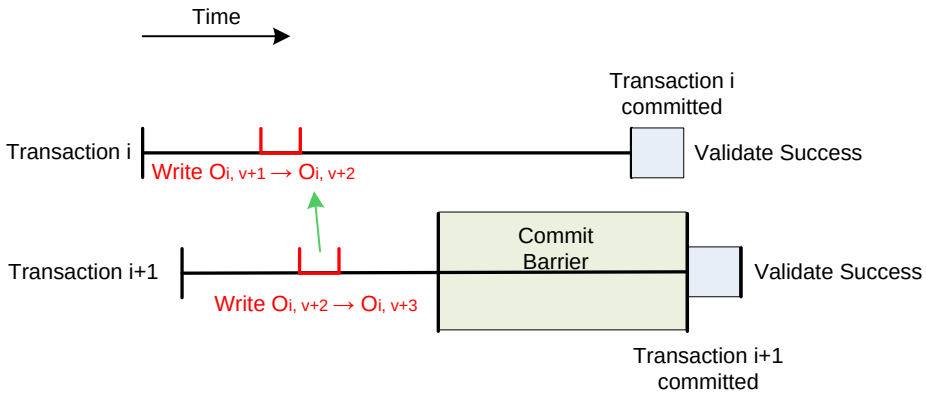
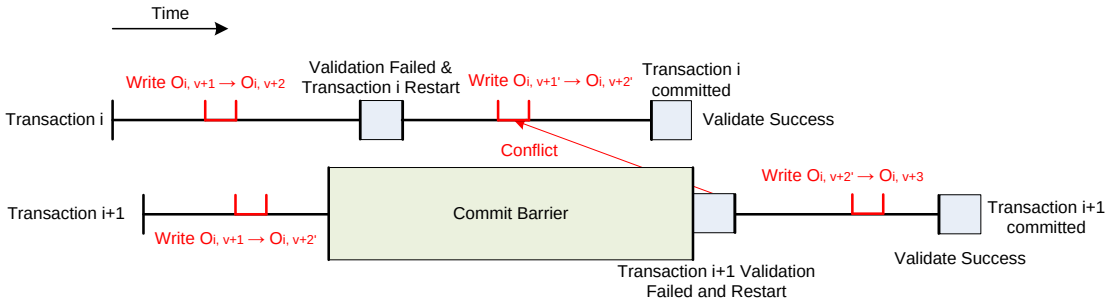
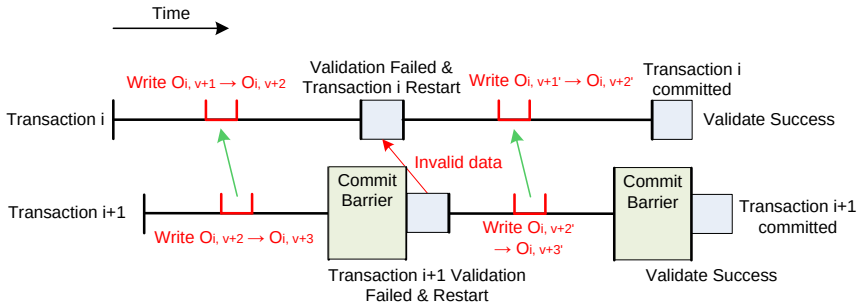


Figure 24 Multi-Version with Commit Barrier

We believe any conflict resolution mechanism must strive to allow transactions to be committed successfully under normal operations. So that, by applying the multi-version approach, the tentative data used in the following transactions (referred to as consumer transactions) will eventually be made permanent so that the consumer transactions can proceed to being committed. This would help increase the system throughput.



(a)



(b)

Figure 25 Multi-Version Speculation with Provider Restart

However, in some cases, the transaction that produced the tentative version (referred to as the provider transaction) may have to be aborted and restarted, and the tentative data would become invalid. This would force the dependent consumer transactions to be rolled back and restarted as well. However, we still can, during the retry of the transactions, use the new tentative data. The performance would still be much better compared with the single version based approach. Figure 25 (a) and (b) show an example with a comparison between the two approaches: (1) When only the commit barrier is enabled and, (2) when both the commit barrier and the multi-version mechanisms are enabled. In Figure 25 (b), we can see that the transaction T_{i+1} still can commit right after T_i. The only different is that T_{i+1} has to rollback and restart due to abort of T_i.

The multi-version approach could make concurrent transaction processing more efficient provided that the tentative data is generated by the right transaction. If the tentative data is from a wrong transaction or it has been re-written to, the consumer transaction would have to be aborted and restarted, as shown in Figure 26. Transaction T_i and T_{i+1} both utilize the tentative data generated by T_{i-1}. T_i may take the advantage of using the tentative version from T_{i-1}. However, transaction T_{i+1} would have to be restarted since the data is overwritten by T_i.

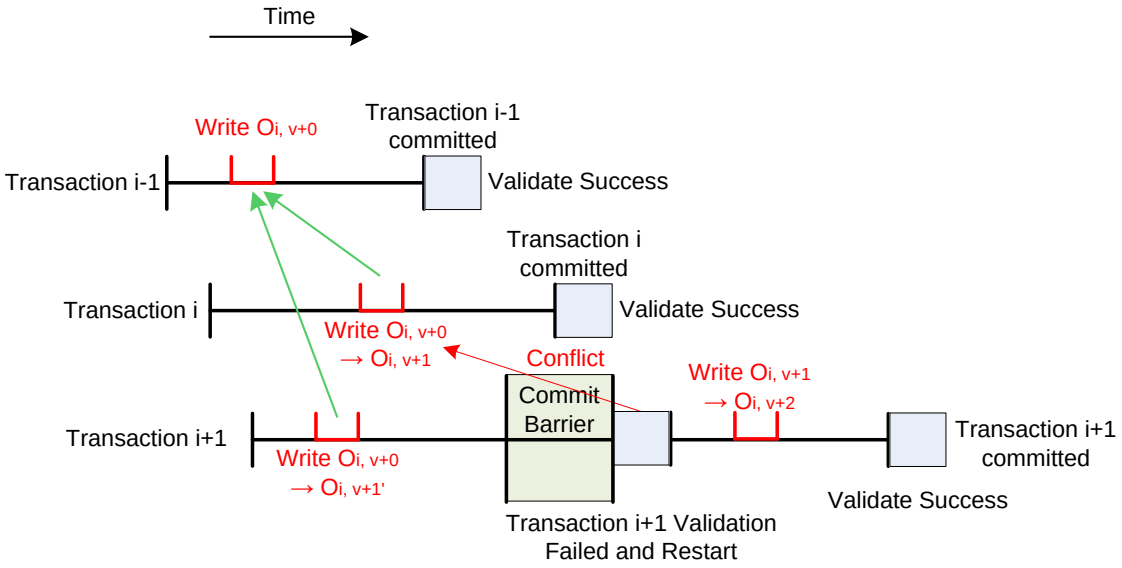


Figure 26 Multi-Version Speculation with Tentative Data Re-written

These cases show the basic rules how the commit barrier and multi-version approach works to solve the conflicts. In the next section, we describe how to implement them in our BFT framework.

6.2.3 Speculative Concurrent BFT

The combination of commit barrier and multi-version speculation, as described in previous sections, can solve the conflicts and enable efficient concurrent transaction processing. We now describe how to implement them in our BFT framework and focus on system wide scenarios.

In practical systems, we may encounter more complicated scenarios than the examples shown before. It is possible that a transaction accesses a data item out of order, such as transaction T_{i-1} arrives later than transaction T_i and both of them reads a shared data item, in which case, the transaction with higher sequence number, transaction T_i here, would have to be aborted and retried as soon as the out-of-order conflicting operation is detected. Figure 27 shows an example of how out-of-order situations are handled by our concurrent BFT framework (denoted as C-BFT) and by a BFT framework with strict sequential execution of all transactions (denoted as S-BFT). Please note that the commit barrier ensures that all transactions commit following a total ordering typically determined based on the order of request arrival. If transactions arrive out of the order or try to commit out of the order, they have to wait until all transactions with lower sequence numbers have been committed.

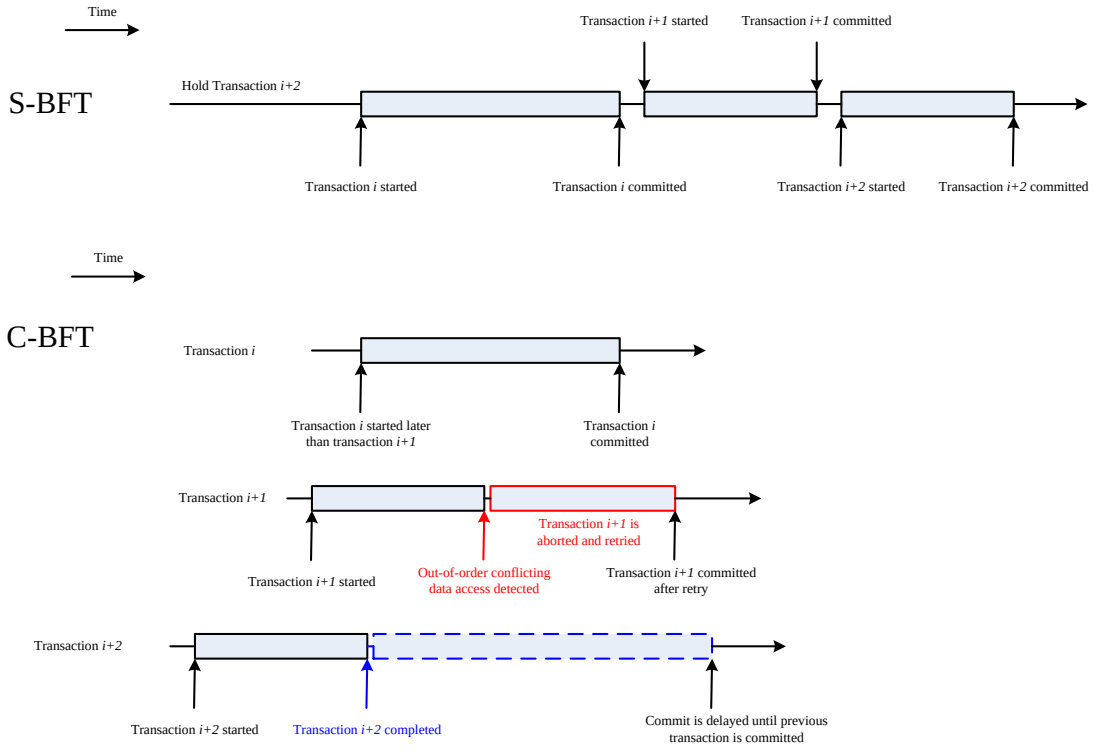


Figure 27 BFT Framework with Strict Sequential Execution of all Transactions (S-BFT) and Concurrent BFT Framework (C-BFT) with an Example of How Out-of-order Situations are Handled

6.3 Concurrent BFT Framework

Our concurrent Byzantine fault tolerance (BFT) framework, as shown in Figure 28, supports client-server applications where the server is constructed with software transactional memory (STM). To take the advantages of separation of agreement and execution [68], we built the agreement agent and application server separately as a standalone cluster, so that only $2f + 1$ server replicas are needed to tolerate up to f

faulty replicas on the application side. The total ordering of the requests from clients is ensured by the UpRight agreement cluster [19]. The application servers, in our implementation, are built on top of the LSA-STM open source library [64] to enable software transaction memory.

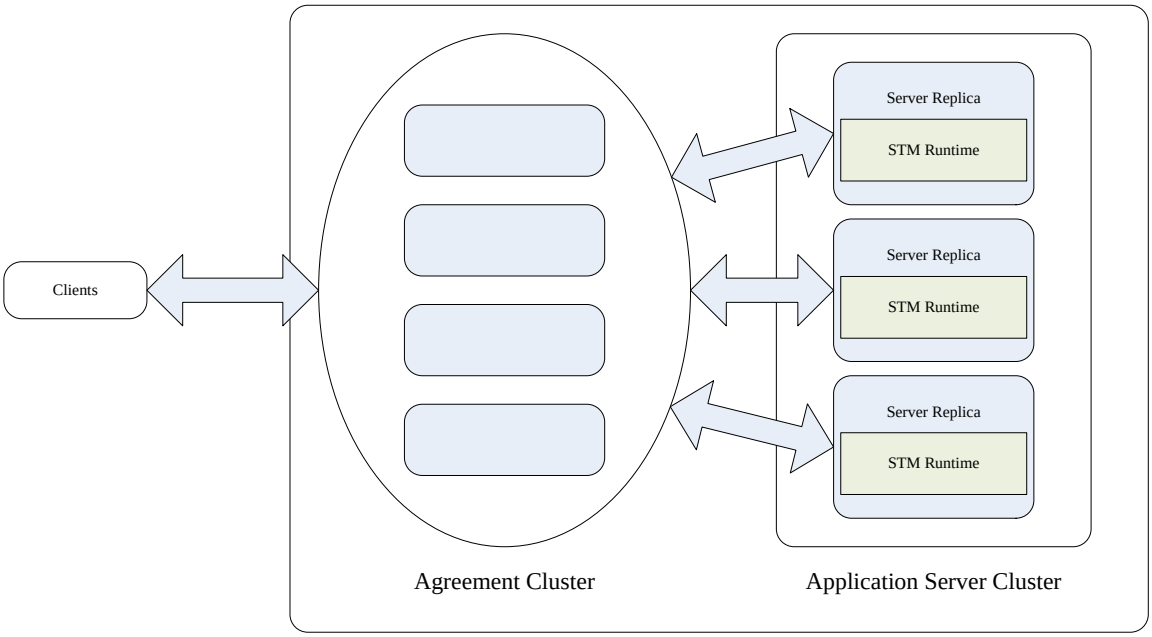


Figure 28 The Proposed Byzantine Fault Tolerance Framework

Client sends their requests to the agreement cluster. And then the agreement cluster totally orders the requests and dispatches ordered requests to the application server replicas. The agreement cluster will be responsible to assign a sequence number to each batch of requests. Hence, the sequence number cannot be directly used on the application server side due to the fact that sequence number is based on batches instead of requests (multiple requests in the same batch will have an identical sequence number). We use a deterministic algorithm to assign a multi-dimensional monotonically increasing timestamp to each request and the corresponding transaction. And this timestamp is then

used to ensure the ordering of each request as well as the transaction triggered by the request.

The batches of requests are disassembled at each server replica. And the request is delivered immediately once it is known that it has been totally ordered. We assume that each request will trigger one and only one transaction at the server replicas. We pre-allocate a thread pool with the size equals to the number of CPU cores. Each thread in the pool will handle a request at a time. Since we have fixed number of threads in the pool, we also build a waiting queue for extra requests. Whenever a thread completes a request, it will fetch the next one in the queue. This approach could significantly increase the system performance for servers equipped with multi-core processors. Figure 29 demonstrates the infrastructure in detail.

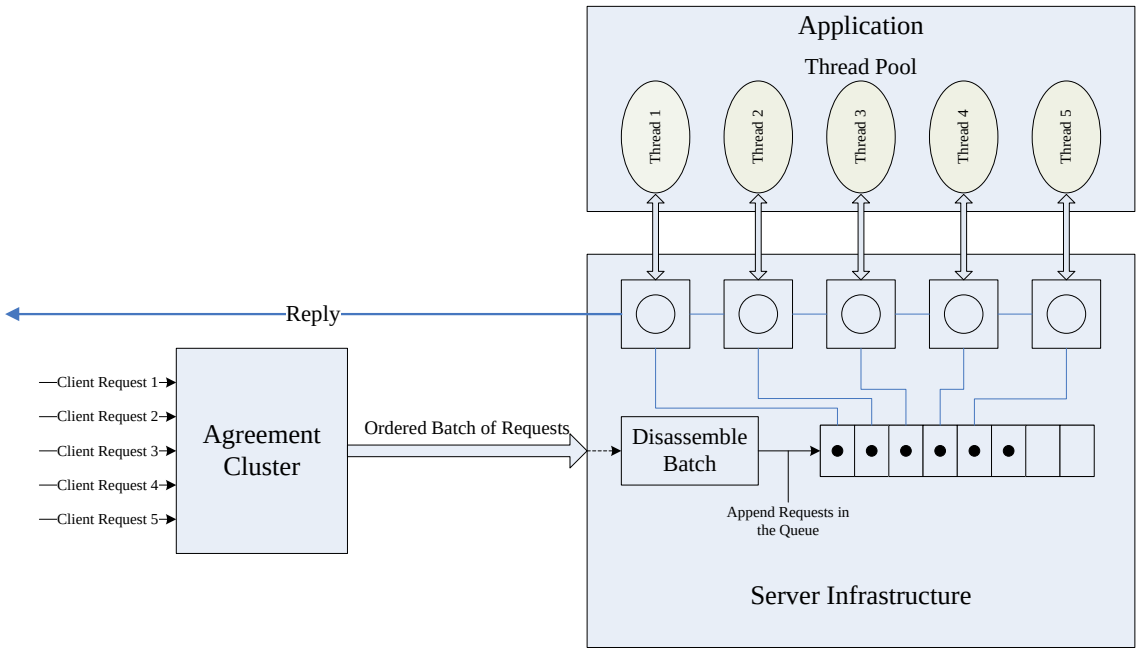


Figure 29 Application Server Infrastructure

6.4 Implementation and Performance Evaluation

The proposed concurrent Byzantine fault tolerance system is implemented in Java. We build our agreement cluster based on the UpRight framework [19] for total ordering the requests from the clients. And on the application server cluster side, we use the LSA-STM library to enable software transactional memory. A comprehensive experimental study has been carried out using our research prototype in a Local-Area Network connected by a Cisco switch. The testbed consists of 14 HP BL460c blade servers and 18 HP ProLiant DL320 G6 servers. Each BL460c server has two Xeon E5405 2.0GHz quad-core processor and 5GB RAM. Each DL320 G6 server is equipped with one Xeon E5620 2.4GHz quad-core processor and 8GB RAM. All servers are running the 64-bit Ubuntu Server Linux operating system.

The basic structure of the test application is a client-server module where the server is supported by our concurrent BFT framework. The agreement server is replicated with $3f + 1$ replicas and the application server is replicated with $2f + 1$ replicas to tolerate up to f faulty replicas in each cluster. Each replica is deployed at a different node in our testbed. In our experiments, we use $f = 1$ because of limited resources, i.e., 3 application server replicas and 4 agreement replicas. All the server replicas are deployed on the BL460c blade server nodes, and the clients are deployed on the DL320 server nodes.

A pre-allocated pool of 8 threads is used to perform concurrent execution. This is to match two quad-core CPUs of each application server replica. The transactions may be aborted and retried; however, it will eventually be committed.

The server maintains a shared data pool with 100 data items, and each transaction accesses 10 data items and perform write operations on them. The data items are selected pseudo-randomly according to a predefined sharing rate. For example, a 20% sharing rate means a transaction will only access 2 items in the shared data pool and another 8 from its private data items. To characterize non-trivial processing load, a finite processing delay is artificially introduced at the server for each transaction in the form of busy loops, i.e., the server executes an empty while loop until the predefined timeout has fired. We use two types of processing load in our experiments: (1) fixed length, and (2) random processing delays with a Poisson distribution.

Furthermore, to explain the performance results, we define some symbols here:

- C-BFT: Concurrent Byzantine fault tolerance system
- S-BFT: Sequential Byzantine fault tolerance system (original BFT system with all requests processed sequentially one after another)
- Fixed-i%: Fixed processing time for each transaction in our BFT framework (C-BFT) with i% data sharing rate
- Poisson-i%: Random processing time with Poisson distribution for each transaction in our BFT framework (C-BFT) with i% data sharing rate

During the first part of the experiments, we set the fixed processing time for 5ms and the Poisson distribution with a mean of 5ms. The following scenarios are shown in Figure 30.

(1) C-BFT (Fixed-i%): Concurrent BFT with 5ms fixed processing time where i varies from 0 to 100 with 20 increment. For comparison purpose, S-BFT with 5ms processing time is included.

(2) C-BFT (Poisson-i%): Concurrent BFT with random processing time with the Poisson distribution with a mean of 5ms. Same as the first test, i changes from 0 to 100 with five equal steps.

The throughput test results are summarized in several figures. Figure 30 shows the average throughput with respect to different number of concurrent clients under various C-BFT Fixed scenarios, and the S-BFT scenarios for comparison. Figure 31 shows the throughput performance with respect to different number of concurrent clients under different C-BFT Poisson scenarios. As expected, the lowest throughput is for the sequential BFT with no concurrent execution and the highest throughput is observed for concurrent BFT with 0 percent data sharing rate, owning to the fact that there is no shared data among transactions. Without shared data, transactions will only work on their own data and it won't cause any conflicts. So the best performance is expected in this scenario. For all other scenarios, the larger sharing rate, the more possibility of getting conflicts during the operations, which leads to a worse throughput.

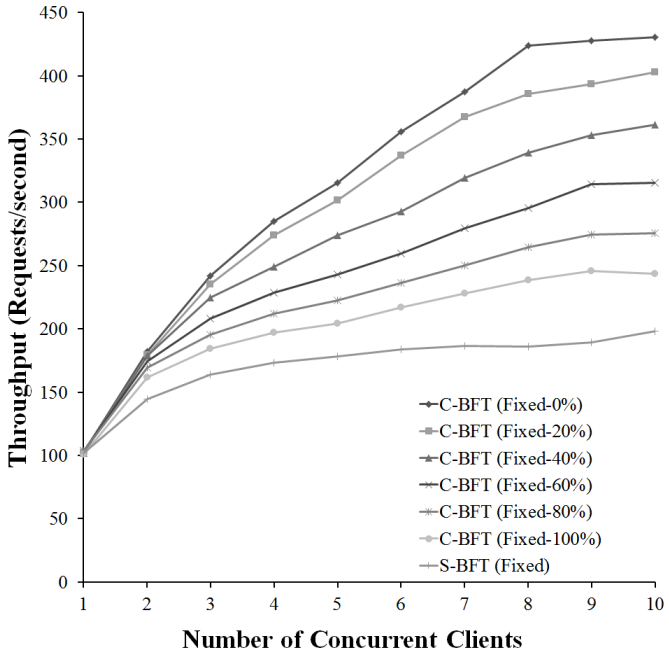


Figure 30 Throughput versus the Number of Concurrent Clients for C-BFT Fixed Configurations

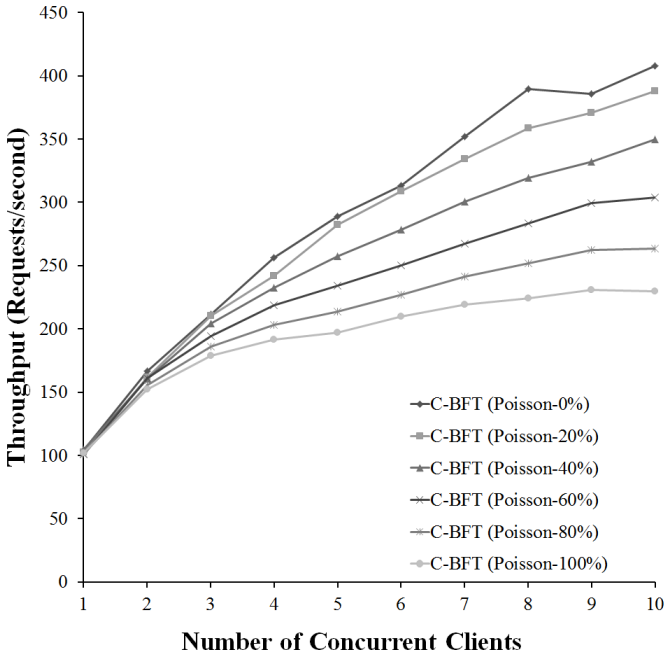


Figure 31 Throughput versus the Number of Concurrent Clients for C-BFT Poisson Configurations.

Figure 32 and Figure 33 show the average and the peak throughput dependency on the data sharing rate for the three sets of scenarios. It can be seen that the throughput decreases with a reasonable amount with larger data sharing rates. We use S-BFT as references in the figure, whose results show as a horizontal line. It makes sense since the data sharing rate has no impact for sequential processing.

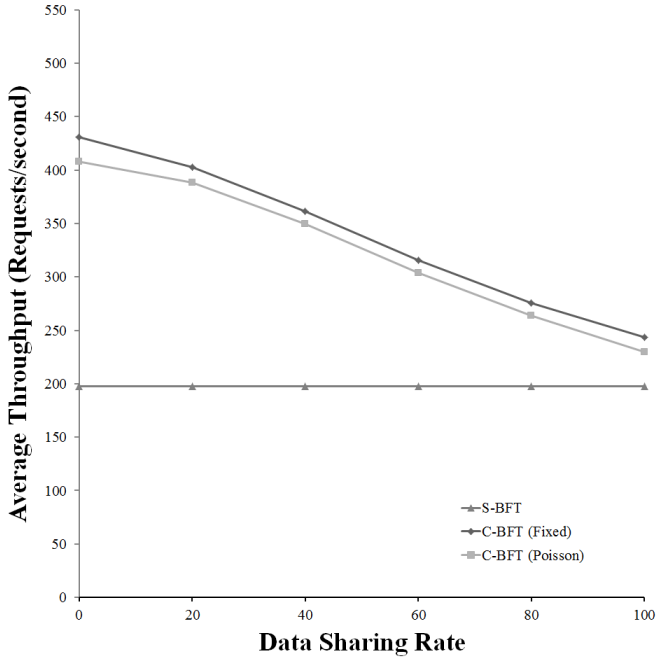


Figure 32 Average Throughput versus Different Data Sharing Rates.

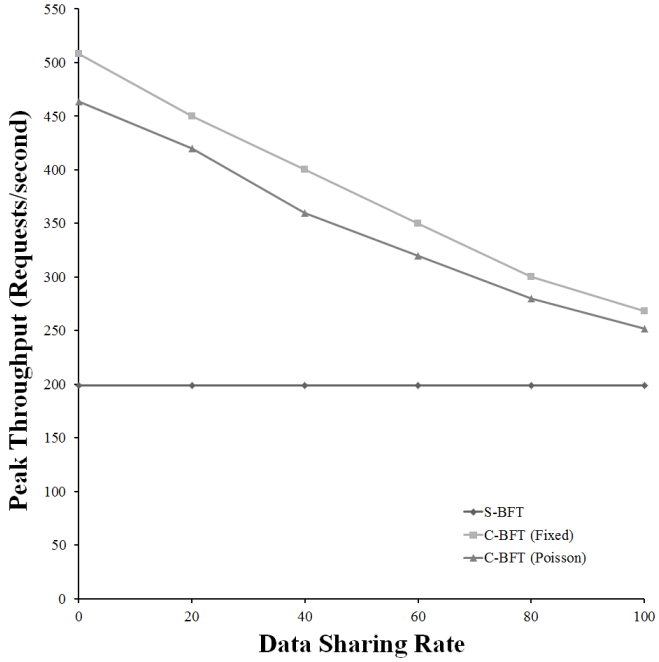


Figure 33 Peak Throughput versus Different Data Sharing Rates.

Figure 34 shows the throughput results with the fixed processing time and Poisson distribution processing time for two scenarios with 0% and 100% sharing rates. As expected, the scenario with the fixed processing time has better performance. The performance of the system with Poisson distribution processing times is worse because the commit barrier causes all transactions to wait for the previous ones to complete. For the fixed processing time situation, the later one can commit immediately if there is no conflicts detected. However, for dynamic processing time, if one transaction takes longer time, all the followings transactions would be impacted, as shown in Figure 35 and Figure 36.

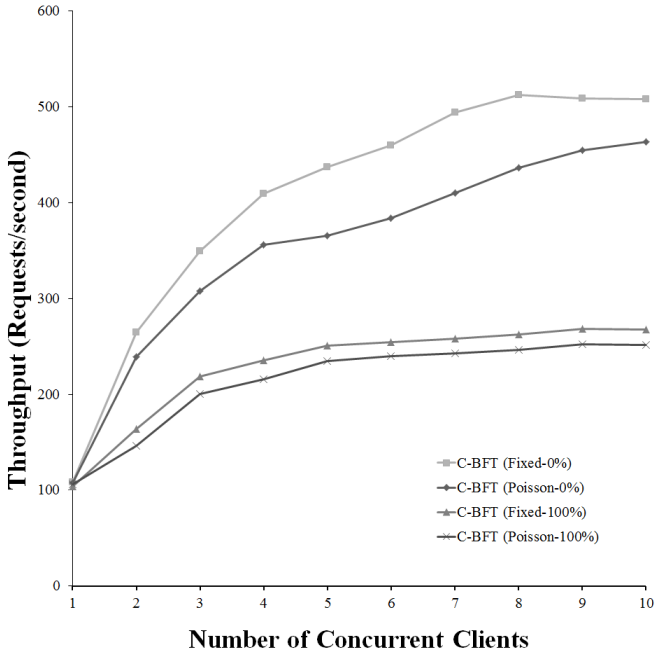


Figure 34 Throughput versus the Number of Concurrent Clients for Comparing Fixed and Poisson Distribution Processing Time.

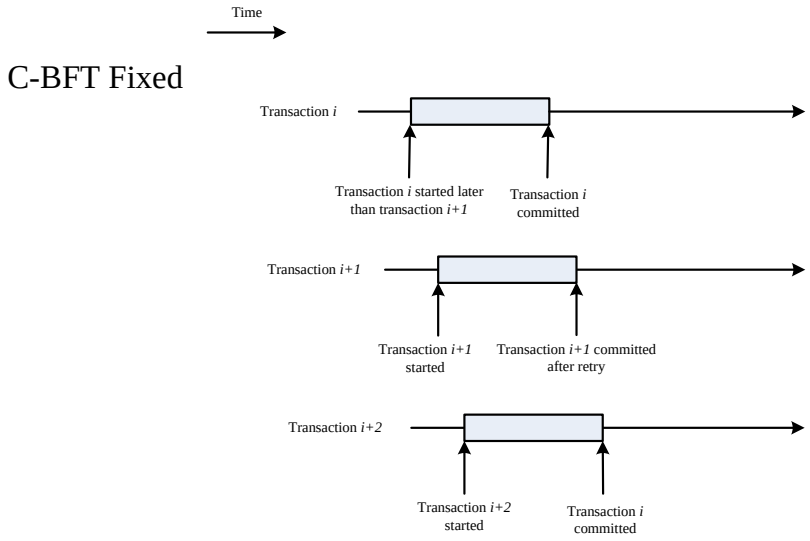


Figure 35 Concurrent BFT with Fixed Processing Time under Normal Operations

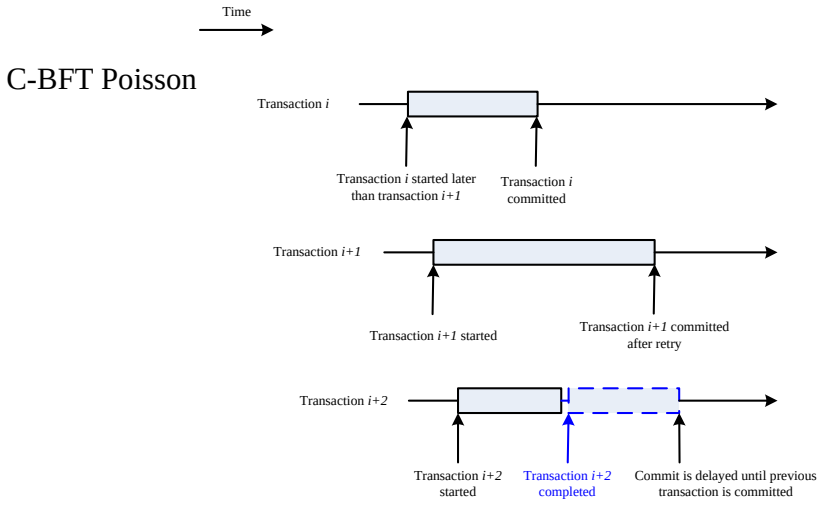


Figure 36 Concurrent BFT with Poisson Distributed Processing Time under Normal Operations

To study the inner workings of the system, we profile the number of conflicts and aborts, in addition to the number of commits in each run. Each of clients sends 100,000 requests consecutively with a pre-defined data sharing rate. We recorded the total number of commits, conflicts and aborts, and then calculated the conflict rate and abort rate. The

profiling results of abort rate and conflict rate for C-BFT fixed scenarios are shown in Figure 37 and Figure 38. And the profiling results for C-BFT Poisson scenarios are shown in Figure 39 and Figure 40. From the figures, we can see that the conflict and abort rates increase exponentially with the number of concurrent clients, and with the sharing rates. This makes sense since both the larger data sharing rate and the more concurrent clients are dedicated more chances of conflicts.

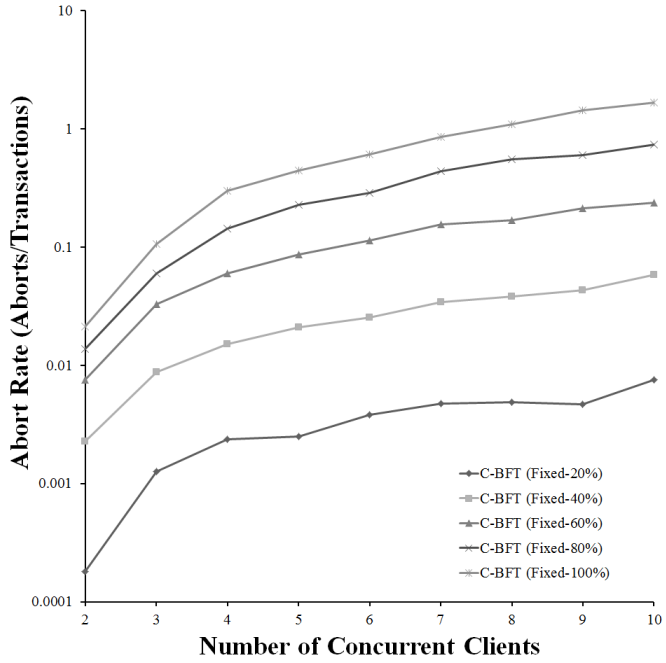


Figure 37 Conflict Rate in Terms of Average Number of Conflicts per Transaction versus Different Number of Concurrent Clients.

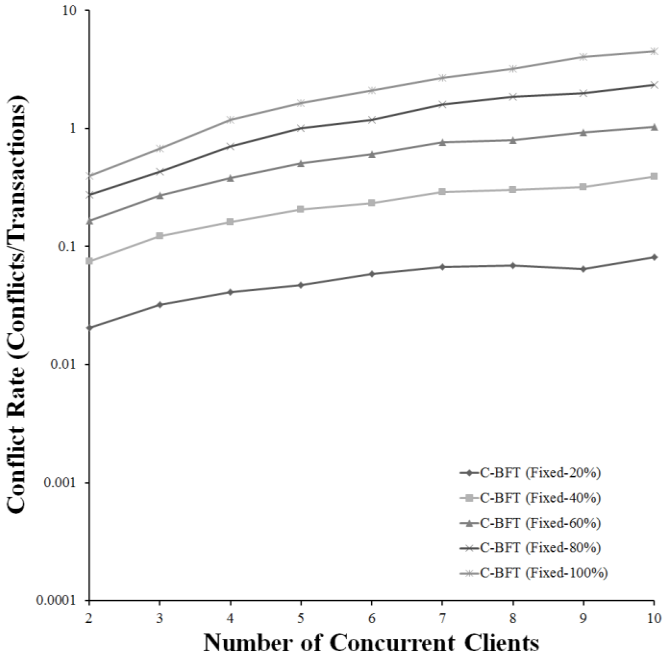


Figure 38 Abort Rate in Terms of Average Number of Aborts per Transaction versus Different Number of Concurrent Clients.

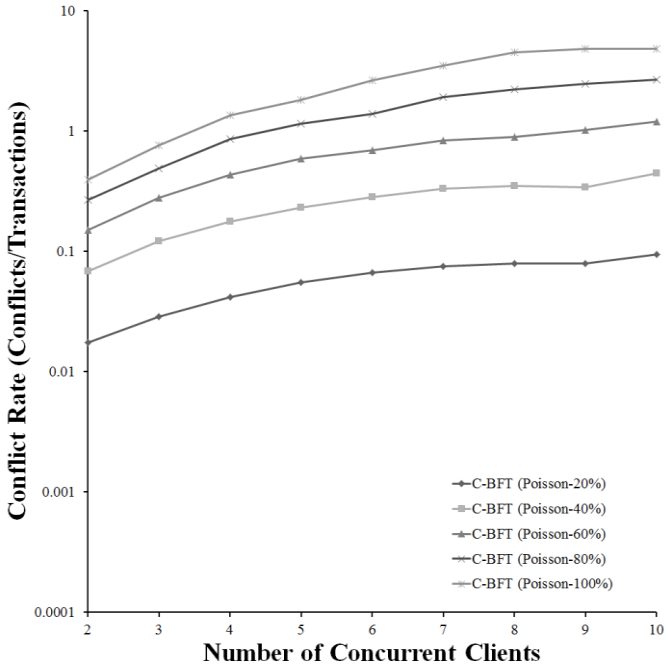


Figure 39 Conflict Rate in Terms of Average Number of Conflicts per Transaction versus Different Number of Concurrent Clients.

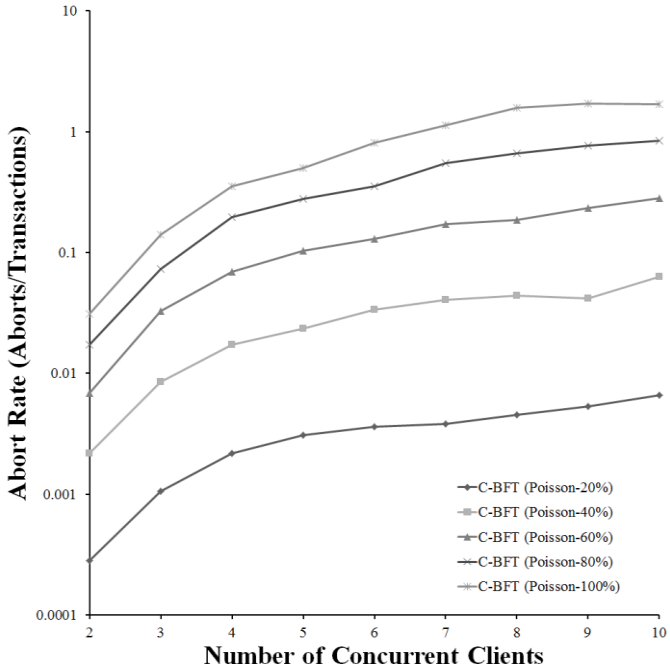


Figure 40 Abort Rate in Terms of Average Number of Aborts per Transaction versus Different Number of Concurrent Clients.

Furthermore, as shown in Figure 41 and Figure 42, the abort rate for dynamic processing time is higher than that for the fixed processing time regardless of sharing rate and number of concurrent clients, which are already explained in Figure 35 and Figure 36.

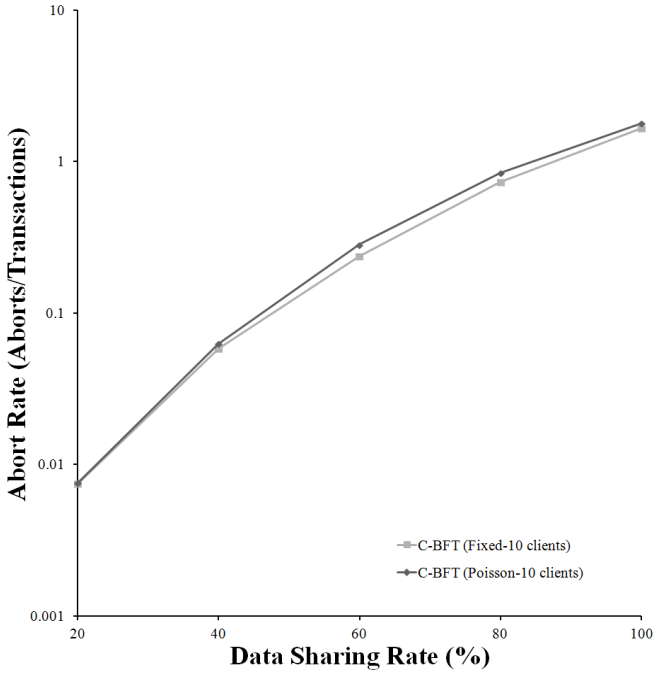


Figure 41 Abort Rates Observed for Different Sharing Rate

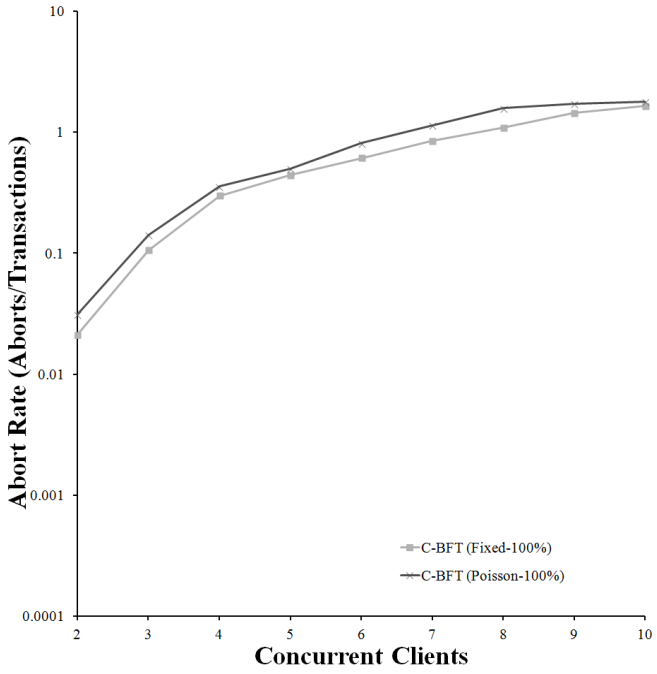


Figure 42 Abort Rates Observed for 10 Concurrent Clients with Different Data Sharing Rates

The test results shown above confirm that indeed the performance is significantly improved with our proposed concurrent BFT system compared with sequential BFT in all circumstances tested. The throughput improvement ranges from 28%, when data sharing rate is 100%, to 125%, when data sharing rate is 0%. From the performance evaluation results, we can make the following conclusions:

(1) Smaller data sharing rates lead to better throughput;

(2) Fixed processing time for each transaction leads to better throughput.

Both can be easily explained. When the data sharing rate gets higher, it is more likely that some transactions will involve conflicting operations and some of the transactions will be aborted and retried. Furthermore, if a transaction is aborted and

retried, all others with higher sequence numbers would have to be delayed, or possibly be aborted and retried also, until the current one is committed eventually. Therefore, the performance of the system with smaller data sharing rate will be better than the one with a larger sharing rate. It also makes sense that the throughput is better when all transactions take similar amount of time to complete. When the processing time to complete a transaction follows the Poisson distribution, the wait-to-commit time will be impacted by a slow transaction. All later transactions would have to wait for the slowest transaction to complete before they can commit. Hence, the performance is reduced. On the other hand, when transactions take the same amount of time to complete, the next one, if there is no conflict, can be committed immediately with minimized overhead.

In Figure 32, it is also interesting to see that the reduction in throughput with more concurrent clients and higher data sharing rates is less than one would have expected. This is because when the aborted transactions are retried, they are still processed concurrently.

6.5 Conclusion

In this chapter, we presented our software transactional memory (STM) based concurrent Byzantine fault tolerance (BFT) framework to maximize the performance by allowing concurrent processing. The strategies are based on two ideas: (1) commit barrier, which is used to commit concurrent transactions according to a assigned total order, and (2) multi-version speculation (works with commit barrier), which allows the

tentative data to be used in later transactions. In essence, the dependencies between concurrent transactions can be discovered and handled dynamically by using the software transactional memory during runtime. If there is no conflict, transactions will be processed concurrently and committed according to the total order of the requests. When conflicts re detected, some transactions may have to be aborted and retried. And eventually, all transactions will be committed successfully. Furthermore, some of the conflicts can in fact be resolved without aborting transactions in the multi-version approach.

A comprehensive performance evaluation of our proposed speculative and concurrent BFT framework is carried out to characterize the effectiveness and limitations. The results show that the overall system performance is significantly increased even in the worst case with every transaction has 100% data from shared data pool. Furthermore, we observed that the throughput not only depends on the data sharing rate, but also the distribution of the processing time.

CHAPTER VII

CONCLUSION AND FUTURE WORK

In this chapter, I summarize my main research contributions in this dissertation and outline some future work. My main contributions include:

- The classification of common types of replica nondeterminism and a set of mechanisms to control replica nondeterminism in the context of Byzantine fault tolerance computing,
- A migration-based proactive service recovery scheme to support long-running Byzantine fault tolerance systems and,
- A set of mechanisms to enable concurrent Byzantine fault tolerant execution of requests based on the software transactional memory model.

The future work will focus on extending my current mechanisms to further reduce the probability of conflicts among concurrent operations and hence facilitate even higher system throughput.

7.1 Conclusion

BFT algorithm is a promising technique to utilize redundancy resources to tolerance Byzantine faults for stateful system. However, the existing BFT algorithms [13, 14, 15, 16, 20, 68] lack the mechanisms to deal with many common types of nondeterministic operations. Such algorithms also require the requests to be executed sequentially to achieve strong replica consistency. Additionally, the assumption of only one-third of the service replicas can be faulty is impossible to hold without additional mechanisms because an adversary would continuously attempt to compromise more replicas over time. To address this concern, several proactive recovery schemes have been proposed [13, 15, 48, 53, 54, 55]. However, they all have the following issues:

- They rely on the rebooting to repair a replica, which may not be effective if hardware components are damaged;
- They may introduce artificial unavailability during a round of proactive recovery;
- They lack mechanisms to coordinate the replicas during a round of proactive replica such that only a small portion of replicas can undergo recovery at any time.

In this dissertation research, we aimed to address all the issues identified above. First, we provided a classification of common types of replica nondeterminism, and introduced a set of mechanisms to handle these types of nondeterminism systematically. If the type of nondeterminism is non-verifiable pre-determinable (NPRE), an extra phase, which we refer to as the pre-prepare-update phase, is used to control the nondeterminism.

The idea is to let every replica contributes its share of nondeterministic values to prevent a faulty replica from dominating the final value, which could compromise the system integrity. For verifiable post-determinable nondeterminism (VPOST) and non-Verifiable post-determinable nondeterminism (NPOST), we have to add a whole round of Byzantine agreement in the post-commit phase to ensure that correct replicas could reach an agreement on the nondeterministic values. Additionally, to prevent the system from crashing, we provisioned a separate monitoring thread for NPOST as governance just in case the main execution thread crashes or hangs.

Second, we presented a service migration based proactive recovery approach. The three issues we pointed out earlier are resolved by the following means: (1) remove the time-consuming recovery step out of the critical path and involve system administrator, if necessary, to fix the problems manually; (2) use a dynamically adjustable service migration interval based on the observed system load and system availability requirements; (3) provide extra resources as standby node pool and use a registration protocol for replica coordination on the membership of standby nodes.

Third, we proposed to use software transaction memory based concurrent execution to lift the limitation of sequential processing and significantly improved the system performance. In our approach, multiple requests are executed concurrently and the commit order is controlled based on the total ordering of incoming requests. Furthermore, multi-version is utilized to pre-execute the requests and hold the result temporarily until the execution is validated. This scheme may significantly reduce the conflict rate of concurrent operations, which is essential to achieve better system throughput.

7.2 Future Work

We have shown that speculative concurrent BFT indeed can significantly improve the performance of the replicated system. However, if we can further classify the write operations, we can make it even better.

The idea hinges on the write operations. We observe that the write operations can different impact based on their relationship with operation history. If we further classify the write operations to history related and history unrelated. It will reduce the possibility of conflicts, which would lead to further performance improvement.

The number of conflicts is the key factor of STM performance. Fewer conflicts, with no doubt, will lead to better throughput. After further classification, even if a history unrelated write operation is followed by any other writes, it won't cause write/write conflict, which will decrease the number of conflicts and hence further improve the performance of whole system. We call it the improved STM solution.

We still use the multi-version based approach similar to LSA-STM [60] and make a tentative value transparent. Later operations can see a tentative value and will use it for its own. When multiple transactions access the same piece of data concurrently, improved STM will resolve the conflicts and guarantee the most important rule of concurrence serializability.

Based on the history relationship, we define two types of write operations, history related write and history unrelated write. The read operations are as usual.

- Read: Based on the sequence number, the read operation should always return the most up-to-date value of the data accessed.
- History Related Write: The write operation relies on the previous data value. This type of operations needs the data values from previous transactions and must be executed in the order as determined by the sequence number. A history related write is always preceded with a read operation on the same data item.
- History Unrelated Write: A write operation simply writes a new value to the data item, regardless of the previous value. A history unrelated write may be executed immediately after the commit barrier, even though there might be unrelated conflicts. Although the value of the previous operation may be overwritten immediately, that operation still have to be carried out since other operations in between may access the data. History unrelated writes can be identified when there is no prior read operation on the same data item in the same transaction.

7.2.1 Conflicts Model Revisited

We have already discussed the conflicts model in section 6.1 and knew that conflicts management plays a very import role during concurrent control. Now let's get into details about how we will take advantage of the classification of the write operations. In the following description, the symbol '/', is used as a separator to indicate which operation happens first with no concern about the sequence number assigned to the

transaction. For example, read/write means read happens before write. This read operation may or may not come with a lower sequence number.

When we have write/write scenarios, we focus on the one with higher sequence number and will take different actions based different types of write operations it.

- History Related Write: If the history related write comes with higher sequence number and happens after the other write operation, the read operation, in read and write pair, will take the tentative value from the cache and use it in the following history related write. As normal transaction, the commit has to wait until all previous transactions have completed. If the write with a lower sequence number is executed later, write/write conflict occurs and the transaction with history related write operation has to be aborted and restarted as shown in Figure 43.

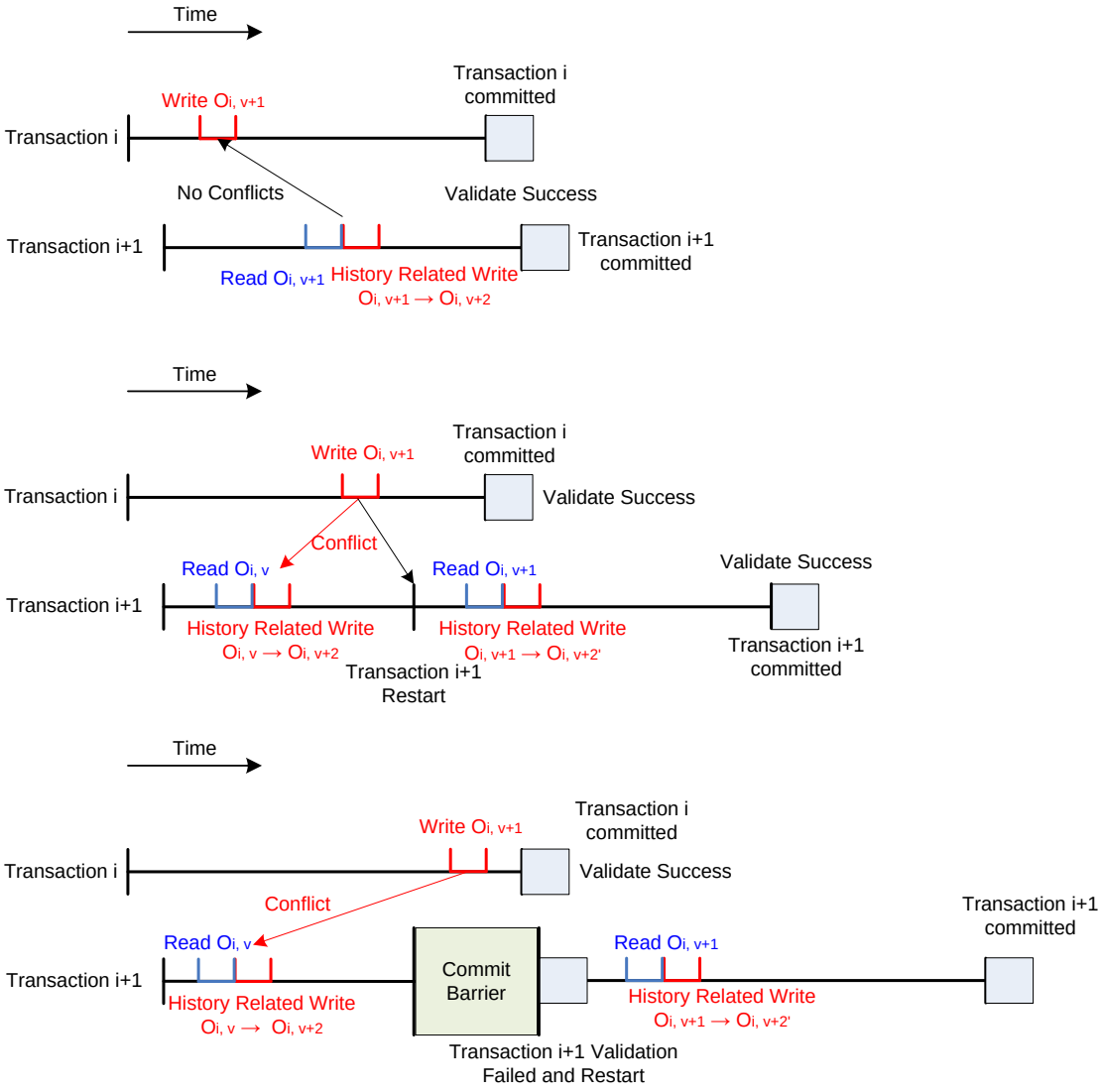


Figure 43 Write/History Related Write

- History Unrelated Write: Two history unrelated writes do not conflict with each other. If a write with a lower sequence number happens first, the history unrelated write can simply re-write with a new version of the data item. On the other hand, if the history unrelated write comes with a larger sequence number but executes first, it will write to the tentative cache value until the validation is succeeded. This is an unrelated conflict which doesn't need to

abort the later transaction. Please note that all transactions with lower sequence numbers that are executed later than the history unrelated write must still be committed first.

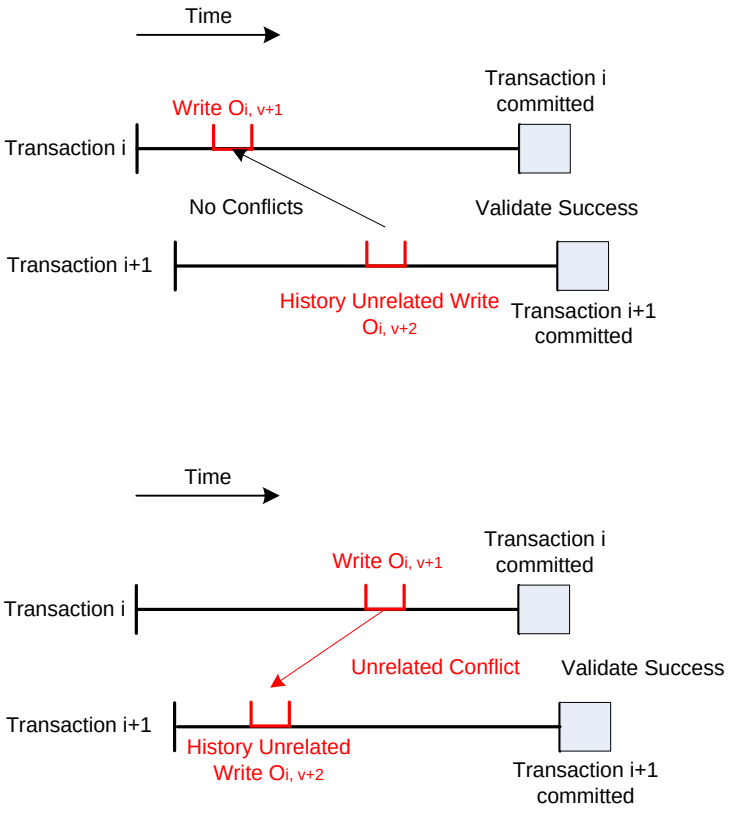


Figure 44 Write/History Unrelated Write

7.2.2 Improved Concurrent Speculative BFT

In previous section, we revisited the conflicts model and outlined a solution that could further reduce the conflicts. By further classifying the write operations, we can see that some conflicts originally exist are gone. Now let's apply this approach to a more complicated case as shown in Figure 45.

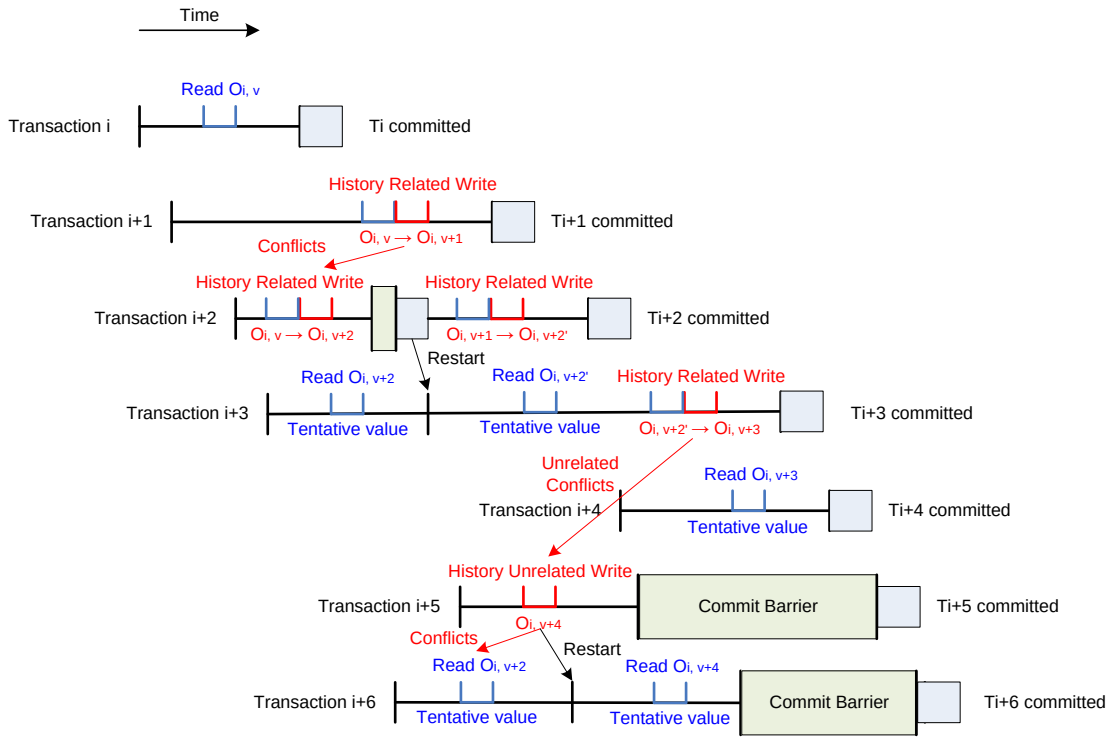


Figure 45 A Complicated Example with Improved Speculative Concurrent BFT

There are 7 current transactions with sequence numbers range from i to $i + 6$. Transaction i only contains a read operation. Transaction $i + 1$ involves a history related write to the data O_i and updates it to version $V + 1$. This transaction could be committed without any problem. Transaction $i + 2$ also performs a history related write on the same data and happens before transaction $i + 1$. It causes a conflict and transaction $i + 2$ have

to be aborted and restarted, which would also affect transaction $i + 3$ since it reads the tentative data from transaction $i + 2$. Both transactions are eventually committed after they are restarted. Transaction $i + 4$ has only a read operation using the tentative value from transaction $i + 3$, and it can be committed successfully. Transaction $i + 5$ gets executed earlier than transaction $i + 4$. Transaction $i + 5$ involves a history unrelated write operation. Although the transaction with a lower sequence number i+3 wrote to the same data later than the transaction with $i + 5$, the unrelated conflict wouldn't cause transaction $i + 5$ abort. The only thing is that transaction $i + 5$ is blocked at the commit barrier until transaction $i + 4$ is committed. The last transaction $i + 6$ came even earlier than $i + 5$, and it reads the invalid data initially, and hence, it must be aborted and restarted. During the re-execution of the transaction, it accessed the correct value. As can be seen, with the write operations further classified, transaction $i + 5$ is committed without having to be restarted.

Although we believe this improved speculative concurrent BFT will further improve the performance. More investigation is necessary to establish its theoretical foundation and to demonstrate its effectiveness for practical applications in the future work.

BIBLIOGRAPHY

[1] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M., and Wylie, J. J.: Fault-scalable Byzantine fault-tolerant services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 39(5): pp. 59-74, Brighton, UK, October 2005.

[2] Arkin, B., Hill, F., Marks, S., Schmid, M., and Walls, T. J.: How we learned to cheat at online poker: A study in software security. http://www.developer.com/java/other/article.php/10936_616221_1, September 1999.

[3] Basile, C., Whisnant, K., and Iyer, R.: A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pp. 149-158, San Francisco, CA, June 2003.

[4] Basile, C., Whisnant, K., Kalbarczyk, Z., and Iyer, R.: Loose synchronization of multithreaded replicas. In *Proceedings of the International Symposium on Reliable Distributed Systems*, pp. 250-255, Suita, Japan, October 2002.

[5] Bressoud, T. and Schneider, F.: Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1): pp. 80-107, February 1996.

[6] Bressoud, T.: TFT: A software system for application-transparent fault tolerance. In *Proceedings of the IEEE 28th International Conference on Fault-Tolerant Computing*, pp. 128-137, Munich, Germany, June 1998.

[7] Brito, A., Fetzer, C., and Felber, P.: Minimizing latency in fault-tolerant distributed stream processing systems. In *The 29th Int'l Conference on Distributed Computing Systems (ICDCS 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2009.

[8] Brito, A., Fetzer, C., and Felber, P.: Multithreading-Enabled Active Replication for Event Stream Processing Operators. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, pp. 22-31, IEEE Computer Society, Washington, DC, USA.

[9] Brito, A., Fetzer, C., Sturzrehm, H., and Felber, P.: Speculative out-of-order event processing with software transaction memory In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pp. 265-275, New York, NY, USA: ACM, 2008.

[10] Cachin, C., Kursawe, K., and Shoup, V.: Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pp. 123-132, June 2000.

[11] Cachin, C., Kursawe, K., Lysyanskaya, A., and Strobl, R.: Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 88-97, Washington, DC, 2002.

[12] Castro, M. and Liskov, B.: Authenticated Byzantine fault tolerance without public-key cryptography. *Technical Report MIT-LCS-TM-589*, MIT, June 1999.

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

[14] Castro, M., and Liskov. B.: Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 173-186, New Orleans, LA, February 1999.

[15] Castro, M., and Liskov. B.: Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, page 19, San Diego, CA, October 2000.

[16] Castro, M., Rodrigues, R., and Liskov. B.: BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3): pp. 236-269, August 2003.

[17] Chai, H., Zhang, H., Zhao, W., Melliar-Smith, P. M., Moser, L. E.: Toward trustworthy coordination for web service business activities. *IEEE Transactions on Services Computing*, 2012. 6(2): pp. 276-288, 2013.

[18] Chen, B. and Morris, R.: Certifying program execution with secure processors. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pp. 133-138, Lihue, HI, May 2003.

[19] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T.: Upright cluster services. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 277-290, 2009.

[20] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L.: Hq replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations*, pp. 177-190, Seattle, WA, November 2006.

[21] Dai, Y., Levitin, G., and Trivedi, K.: Performance and reliability of tree-structured grid services considering data dependence and failure correlation. *IEEE Transactions on Computers*, 56(7): pp. 925-936, July 2007.

[22] Dai, Y., Pan, Y., and Zou, X.: A hierarchical modeling and analysis for grid service reliability. *IEEE Transactions on Computers*, 56(5): pp. 681-691, 2007.

[23] Dai, Y., Xie, M., and Poh, K.: Modeling and analysis of correlated software failures of multiple types. *IEEE Transactions on Reliability*, 54(1): pp. 100-106, 2005.

[24] Dai, Y., Xie, M., and Wang, X.: Heuristic algorithm for reliability modeling and analysis of grid systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 37(2): pp. 189-200, 2007.

[25] Dai, Y., Xie, M., Long, Q., and Ng, S.: Uncertainty analysis in software reliability modeling by bayesian analysis with maximum- entropy principle. *IEEE Transactions on Software Engineering*, 33(11): pp. 781-795, 2007.

[26] Defago, X., Schiper, A., and Sergent, N.: Semi-passive replication, In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pp. 43-50, 1998.

[27] Dieter, W. R. and Lumpp, J. E.: User-level checkpointing for LinuxThreads programs. In *Proceedings of the USENIX Technical Conference*, Boston, Massachusetts, June 2001.

[28] Distler, T. and Kapitza, R.: Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency. *Proceedings of the sixth Eurosys conference,* pp. 91-106, 2011.

[29] Fischer, M., Lynch, N., and Paterson, M.: Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2): pp. 374-382, April 1985.

[30] http://en.wikipedia.org/wiki/Determinism

[31] Jimenez-Peris, R., Patino-Martinez, M. and Arevalo, S.: Deterministic scheduling for transactional multithreaded replicas. In *Proceedings of the IEEE 19th Symposium on Reliable Distributed Systems*, pp. 164-173, Nurnberg, Germany, October 2000.

[32] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E.: Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of ACM Symposium on Operating System Principles (SOSP),* pp. 45-58, New York, NY, October 2007.

[33] Kotla, R. and Dahlin, M.: High throughput byzantine fault tolerance. *Proceedings of International Conference on Dependable Systems and Networks*, pp. 575-584, June 28 –July 1, 2004.

[34] Lamport, L., Shostak, R., and Pease, M.: The Byzantine general's problem, *ACM Transactions on Programming Languages and Systems*, 4(3): pp. 382-401, 1982.

[35] Lamport, L.: Paxos made simple. *ACM SIGACT News (Distributed Computing Column),* 32: pp. 18-25, 2001.

[36] Malek, M., Polze, A., and Werner, W.: A framework for responsive parallel computing in network-based systems. In *Proceedings of International Workshop on Advanced Parallel Processing Technologies*, pp. 335-343, Bejing, China, September 1995.

[37] Marsh, M. A. and Schneider, F. B.: Codex: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1): pp. 34-47, January 2004.

[38] Martin, J. and Alvisi, L.: Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3): pp. 202-215, July-September 2006.

[39] Moser, L. and Melliar-Smith, M.: Consistent asynchronous checkpointing of multithreaded application programs based on semi-active or passive replication. US Patent Application No. 20050034014, 2005.

[40] Moser, L. and Melliar-Smith, M.: Transparent consistent semi-active and passive replication of multithreaded application programs. US Patent Application No. 20040078618, 2004.

[41] Narasimhan, P., Moser, L. E., and Melliar-Smith, P. M.: Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *Proceedings of the IEEE 18th Symposium on Reliable Distributed Systems*, pp. 263-273, Lausanne, Switzerland, October 1999.

[42] Ostrovsky, R. and Yung, M.: How to withstand mobile virus attacks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 51-59, Montreal, Quebec, Canada, 1991.

[43] Polze, A., Schwarz, J., and Malek, M.: Automatic generation of fault-tolerant corba-services. In *Proceedings of Technology of Object-Oriented Languages and Systems*, pp. 205-213, Santa Barbara, CA, 2000. IEEE Computer Society Press.

[44] Powell, D.: Delta-4: A Generic Architecture for Dependable Distributed Computing. Springer-Verlag, 1991.

[45] Reiser, H. P. and Kapitza, R.: Hypervisor-based efficient proactive recovery. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 83-92, 2007.

[46] Reiser, H., Domaschka, J., Hauck, F., Kapitza, R., and Schroder-Preikschat, W.: Consistent replication of multithreaded distributed objects. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pp. 257-266, Leeds, UK, October 2006.

[47] Riegel, T., Felber, P., and Fetzer, C.: A lazy snapshot algorithm with eager validation. *Proceedings of the 20th International Symposium on Distributed Computing*, pp. 284-298, 2006.

[48] Rodrigues, R. and Liskov, B.: Byzantine fault tolerance in long-lived systems. In *Proceedings of the 2nd Workshop on Future Directions in Distributed Computing*, June 2004.

[49] Schneider, F.: Implementing fault-tolerant services using the state machine approach: A tutorial, *ACM Computing Surveys*, 22(4): pp. 299-319, 1990.

[50] Singh, A., Maniatis, P., Druschel, P. and Roscoe, T.: Conflict-free quorum-based BFT protocols. *Technical Report 2007-1, Max Planck Institute for Software Systems*, August 2007.

[51] Slember, J. and Narasimhan, P.: Living with nondeterminism in replicated middleware applications. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference*, pp. 81-100, Melbourne, Australia, 2006.

[52] Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P.: Resilient intrusion tolerance through proactive and reactive recovery. In *Proceedings of the IEEE Pacific Rim Dependable Computing Conference*, pp. 373-380, 2007.

[53] Sousa, P., Neves, N. F., and Verissimo, P.: Proactive resilience through architectural hybridization. In *ACM Symposium on Applied Computing*, pp. 686-690, Dijon, France, 2006.

[54] Sousa, P., Neves, N. F., Verissimo, P., and Sanders, W. H.: Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 71-82, October 2006.

[55] Vaughan-Nichols, S. J.: Virtualization sparks security concerns. *Computer*, 41(8): pp. 13-15, August 2008.

[56] Viega, J. and McGraw, G.: Building Secure Software. Addison-Wesley, 2002.

[57] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L. and Dahlin, M.: Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 253-267, Bolton Landing, NY, USA, 2003.

[58] Young, A. and Yung, M.: Malicious Cryptography: Exposing Cryptovirology. Wiley Publishing, 2004.

[59] Zhang, H. and Zhao, W.: Concurrent Byzantine Fault Tolerance for Software-Transactional-Memory Based Applications. *International Journal of Future Computer and Communication*, 1(1): pp. 47-50, 2012.

[60] Zhang, H., Chai, H., Zhao, W., Melliar-Smith, P. M., and Moser, L. E.: Trustworthy coordination for web service atomic transactions. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[61] Zhang, H., Zhao, W., Moser, L. E., and Melliar-Smith, P. M.: Design and Implementation of a Byzantine Fault Tolerance Framework for Non-Deterministic Applications. *IET Software*, 5(3): pp. 342-356, 2011.

[62] Zhao, W. and Zhang, H.: Byzantine Fault Tolerant Coordination for Web Services Business Activities. *Proceedings of the IEEE International Conference on Services Computing*, pp. 407-414, Honolulu, Hawaii, July 8-11, 2008.

[63] Zhao, W. and Zhang, H.: Proactive Service Migration for Long-Running Byzantine Fault Tolerant Systems. *IET Software*, 3(2): pp. 154-164, April 2009.

[64] Zhao, W., Melliar-Smith, P. M., and Moser, L. E.: Fault tolerance middleware for cloud computing, *Proceedings of the IEEE Cloud Computing*, pp. 67-74, Miami, FL, USA, July 2010.

[65] Zhao, W., Moser, L. E., and Melliar-Smith, P. M.: Deterministic scheduling for multithreaded replicas. In *Proceedings of the IEEE International Workshop on Object-oriented Real-time Dependable Systems*, pp. 74-81, Sedona, Arizona, February 2005.

[66] Zhao, W., Moser, L. E., and Melliar-Smith, P. M.: Transparent fault tolerance for distributed and networked applications, *In Encyclopedia of Information Science and Technology*, pp. 1190-1197, Idea Group Publishing, 2005.

[67] Zhao, W.: Byzantine fault tolerance for nondeterministic applications. In *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pp. 74-81, Loyola College Graduate Center, Columbia, MD, USA, September 2007.

[68] Zhou, L., Schneider, F., and Renesse, R. van: Coca: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4): pp. 329-368, November 2002.