2008

Byzantine Fault Tolerance for Nondeterministic Applications

Bo Chen
Cleveland State University

BYZANTINE FAULT TOLERANCE FOR NONDETERMINISTIC APPLICATIONS

BO CHEN

Bachelor of Science in Computer Engineering

Nanjing University of Science & Technology, China

July, 2006

Submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE IN COMPUTER ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

December, 2008

The thesis has been approved

for the Department of **ELECTRICAL AND COMPUTER ENGINEERING**

and the College of Graduate Studies by

_____

Thesis Committee Chairperson, Dr. Wenbing Zhao

_____

Department/Date

_____

Dr. Yongjian Fu

_____

Department/Date

_____

Dr. Ye Zhu

_____

Department/Date

ACKNOWLEDGEMENT

First, I must thank my thesis supervisor, Dr. Wenbing Zhao, for his patience, careful thought, insightful commence and constant support during my two years graduate study and my career. I feel very fortunate for having had the chance to work closely with him and this thesis is as much a product of his guidance as it is of my effort.

The other member of my thesis committee, Dr. Yongjian Fu and Dr. Ye Zhu suggested many important improvements to this thesis and interesting directions for future work. I greatly appreciate their suggestions.

It has been a pleasure to be a graduate student in Secure and Dependable System Group. I want to thank all the group members: Honglei Zhang and Hua Chai for the discussion we had. I also wish to thank Dr. Fuqin Xiong for his help in advising the details of the thesis submission process.

I am grateful to my parents for their support and understanding over the years, especially in the month leading up to this thesis.

Above all, I want to thank all my friends who made my life great when I was preparing and writing this thesis.

BYZANTINE FAULT TOLERANCE FOR NONDETERMINISTIC APPLICATIONS

BO CHEN

ABSTRACT

The growing reliance on online services accessible on the Internet demands highly reliable system that would not be interrupted when encountering faults. A number of Byzantine fault tolerance (BFT) algorithms have been developed to mask the most complicated type of faults — Byzantine faults such as software bugs, operator mistakes, and malicious attacks, which are usually the major cause of service interruptions. However, it is often difficult to apply these algorithms to practical applications because such applications often exhibit sophisticated non-deterministic behaviors that the existing BFT algorithms could not cope with.

In this thesis, we propose a classification of common types of replica non-determinism with respect to the requirement of achieving Byzantine fault tolerance, and describe the design and implementation of the core mechanisms necessary to handle such replica nondeterminism within a Byzantine fault tolerance framework. In addition, we evaluated the performance of our BFT library, referred to as ND-BFT using both a micro-benchmark application and a more realistic online porker game application. The performance results show that the replicated online poker game performs approximately 13% slower than its nonreplicated counterpart in the presence of small number of players.

Keywords: Byzantine fault tolerance, replica nondeterminism, security, replica consistency, replication, intrusion tolerance, performance, online poker game

TABLE OF CONTENTS

LIST OF FIGURES

ACRONYM

BFT Byzantine Fault Tolerance

ND-BFT Nondeterminism Byzantine Fault Tolerance

VPRE Verifiable Pre-Determinable Nondeterminism

NPRE Non-Verifiable Pre-Determinable Nondeterminism

VPOST Verifiable Post-Determinable Nondeterminism

NPOST Non-Verifiable Post-Determinable Nondeterminism

CHAPTER I

INTRODUCTION

The society is increasingly dependent on services provided by computer systems and our vulnerability to computer failures is growing as a result: we expect to have highly-available systems or applications that should work correctly and provide services without interruptions. This requires the system or the application to be carefully designed and implemented, and rigorously tested. However, considering the intense pressure for short development cycles and the widespread use of commercial-off-the-shelf software components, it is not surprising that software systems are notoriously imperfect. Problems such as software crashing, leaking of confidential information, modify or deleting of critical data, or injecting of erroneous information into the application data. These malicious faults often referred as Byzantine faults. The Byzantine faults can be handled by replicating the server and employing a Byzantine fault tolerance (BFT) algorithm as described in [2, 8, 17, 18].

Byzantine fault tolerance algorithms require the replicas to operate deterministically, i.e., given the same input under the same state, all replicas produce

the same output and transit to the same state. However, it is incorrect to assume that practical applications will operate deterministically. Moreover it is equally incorrect to categorize the determinism into a single type. Therefore, when a practical application is replicated to tolerate Byzantine fault, its replica nondeterminism must be analyzed carefully and be tackled properly to ensure replica consistency.

In previous research, although the replica nondeterminism issue has been studied, it is limited to only the most simplistic forms of nondeterminisim, which we term as nondeterminism and verifiable pre-determinable nondeterminism[2, 8, 17, 18]. The former assumes that any nondeterministic operations and their side effects can be mapped into some pre-specified abstract operations and state, which are deterministic. The later assumes that any nondeterministic values can be determined prior to the execution of a request, and such values proposed by one replica can be verified by other replicas in a deterministic manner, and the values are accepted only if they are believed to be correct.

Therefore, new techniques must be carried out to cope with replicated applications that exhibit other types of nondeterministic behavior to guarantee replica consistency. For example, many online gaming applications contain some kind of nondeterminism whose value [4, 14] (e.g., random numbers that determine the state of the applications) proposed by one replica cannot be verified by another one. It is incorrect to treat this type of replica nondeterminism the same as the verifiable pre-determinable nondeterminism because a faulty replica could use a predictable algorithm to update its internal state and collude with its clients, without being detected, which defeats the purpose of Byzantine fault tolerance. As another example,

multi-threaded applications may exhibit nondeterminism whose values [13] (e.g., thread interleaving) cannot be determined prior to the execution of a request (without losing concurrency) which cannot be handled by existing BFT mechanisms.

1.1 Contribution

In this thesis, we introduce a classification of common types of replica non-determinism present in many applications. We propose a set of mechanisms that can be used to control these types of nondeterministic operations. We also describe the implementation of the core mechanisms and their integration with a well-known BFT framework [18]. More, specifically, we make the following research contributions:

- We provide two types of motivating applications to illustrate the inadequacy of current approaches to the problem of replica non-determinism
- We provide a classification of common types of replica nondeterminism for both Byzantine fault tolerance and benign fault tolerance.
- We propose a unified framework to ensure consistent Byzantine fault tolerant replication for applications exhibiting the nondeterministic behavior we have classified.
- We provide a preliminary implementation of the unified framework based on the original BFT framework and report the performance evaluation results of our prototype on handling different types of replica non-determinism.
- We propose a alternative technology with better security result, however, the performance of this technology is not as good as ND-BFT, thus there is still a lot of future research to do on this topic.

1.2 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides background information. We start by describing BFT and other related techniques that used to tolerate Byzantine fault, which is a big picture of what is Byzantine fault and how to tolerant it. Chapter 3 describes ND-BFT: we explain the limitation of original BFT, and provide a systematic classification of different type of replica nondeterminism. The reminder of this chapter describes the corresponding solution for each type of replica nondeterminism and the proof of correctness. The implementation of the ND-BFT library, interface and online poker game that equipped ND-BFT library is presented in Chapter 4. The detailed performance analysis for the ND-BFT library and online poker game is described in the second half of Chapter 4. Chapter 5 discusses related work. Finally, our conclusions and some direction for future work appear on Chapter 6.

CHAPTER II

BACKGROUND

In this chapter, we present the background information including fault tolerance, Byzantine fault tolerance and other Byzantine fault-tolerant techniques to provide a big picture of the importance for a distributed system to obtain such capabilities to tolerate Byzantine fault.

2.1 Fault Tolerance

In this section, we present the basic concept of fault tolerance to show the importance for a distributed system to obtain such capability.

Fault tolerance, an important subject in distributed system design, is defined as a capability that a system can mask the occurrence and recovery from failures. In other words, a fault tolerant system can continue to operate without notice by outside in the presence of failure.

A characteristic feature of distributed systems that distinguishes them from single-machine system is the notion of partial failure. A partial failure may happen

when one component in a distributed system fails. The failure may affect the proper operation of other components, while at the same time leaving yet other components totally unaffected. In contrast, a failure in non-distributed systems is often going to affect all components, and may easily bring down the entire applications.

An important goal in distributed systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously degrade the overall performance. In particular, whenever a failure occurs, the distributed system should continue to operate in an acceptable way whiles repairs are being made, that is, it should tolerate faults and continue to operate to some extent.

There are several types of failure exist during the operation as following:

- Crash failure: A process simply halts
- Omission failure: A process does not respond to incoming requests.
- Timing failure: A process responds too sooner or too later to a request.
- Response failure: A process responds a request in a wrong way.
- Byzantine failure: A process exhibits any kind of failure.

Redundancy is the essence to achieve fault tolerance. When applied to processes, the notion of process group becomes important. A process group consists of several processes that closely cooperate to provide a service. In fault tolerant process groups, one or more processes can fail without affecting the availability of the services. Often, it is necessary that communication within the group be highly reliable, and adheres to stringent ordering and atomicity properties to achieve fault tolerance which is often referred as reliable group communication, or reliable multi-casting.

2.2 Byzantine Fault Tolerance

2.2.1 Byzantine Fault

A Byzantine fault is an arbitrary fault that occurs during the operation by a distributed system. When a Byzantine failure occurs, the system may respond in any unpredictable way which exhibits in real world environment as computers and networks behaves in unexpected ways due to hardware failures, software errors, network congestion and disconnection, as well as malicious attacks. Those problems become increasing crucial nowadays,. because people are increasingly depending on online services.

The term “Byzantine faults” was originated from the classic Byzantine General's problem[1], which several legions lead by one commander and several lieutenants camped outside of the enemy’s castle and wait for commander's command. To make sure each lieutenant gets the same command, each lieutenant is required to send received command (attack or retreat) to the rest of the lieutenants. However, there are one or more traitors; the traitor can be either lieutenant or commander himself that they try to confuse other loyal lieutenants by sending different commands to them. In that case, a loyal lieutenant may receive conflict command and confuse about which one is true. And the campaign would be defeated if the majority of the troops do not follow the same command. This problem can be solved by the Byzantine Agreement: if there is one traitor, we need at least four generals including one commander to make an agreement among the loyal generals. If we using this solution in computer world, we can have following conclusion: to

tolerate f Byzantine fault, we need 3f+1 replicas. Figure 1 shows the proof of this algorithm that for a single Byzantine fault, 4 replicas are needed. (a) If the commander (i.e., primary replica) is faulty, he may send conflicting information to its lieutenants (i.e., replica replicas). However, the lieutenants can exchange information regarding what they heard from the commander and reach the correct decision (attack) based on majority voting. (b) On the other hand, if a lieutenant is faulty, he may lie to other lieutenants regarding the information he has heard from the commander. Other lieutenants can still reach a correct decision based on majority voting. Reducing the number of replicas to 3 cannot guarantee an agreement among the correct replicas.

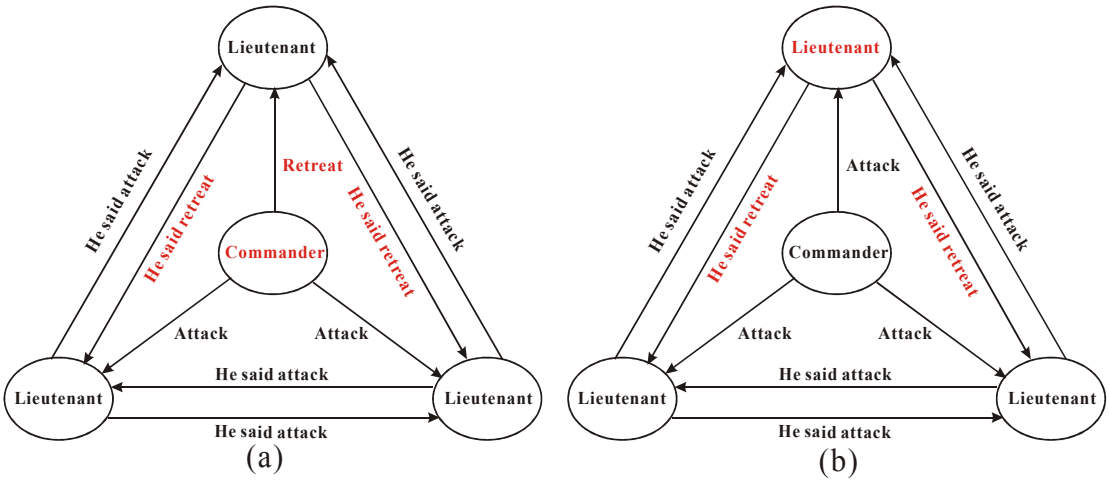

Figure 1: Byzantine Agreement (one traitor)

2.2.2 Byzantine Fault Tolerance

Byzantine fault tolerance [2, 8, 17, 18], a technique that is able to defend against the Byzantine fault. A Byzantine fault tolerant system can reach the same group decision regardless of the existence of Byzantine faulty replicas.

Since distributed applications are often structured in terms of clients and servers, each service comprises one or more servers and executes the clients' request. The state machine replication technique is a general approach to build a fault-tolerant system by replicating the servers and making them to behave identically. The replicated servers coordinate the original server to reach an agreement to tolerate faults. However, it is not enough for this approach to tolerate complicate Byzantine fault.

Therefore, systems that provide critical services must behave correctly in the face of Byzantine faults. Correct services in the presence of failures are achieved through replications: the services runs t a number replicated servers and as more than a third of the servers are non-faulty, the group as a whole continues to behave correct.

Byzantine fault tolerance algorithm, which initial by Castro and Liskov[8], is state machine based protocol. A Byzantine faulty replica may use all kinds of strategies to prevent the normal operations of the replicated services. In particular, it might propagate conflicting information to other replicas or components that it interacts with. To tolerate f Byzantine faulty replicas in an asynchronous environment, we need to have at least 3f+1 number of replicas. An asynchronous environment is one that has no bound on processing times, communication delays, and clock skews. Internet applications are often modeled as asynchronous systems. Usually, one server is designated as primary and the rest are replicas. The protocol move through a series of views, each view is denoted by a view number. The primary for a given view is determined based on the view number. Replicas remain in the current view unless the primary is suspected of being faulty. If the primary behaves in an incorrect or timely

way, the other replicas will execute a view change, selecting a new primary by internal vote and incrementing the view number and moving to a new view.

BFT algorithm has three communication rounds which is referred as three-phase protocol in normal case operation as following:

Pre-Prepare Invoked by the primary after receiving the request from the client that it assigns a sequence number, view number and correspond authenticator and multicast the PRE_PREPARE message to all replicas.

Prepare A replica broadcast the Prepare message to the rest of replicas after it accepts the PRE_PREPARE message.

Commit Once a replica receive 2f+1 PREPARE message that has the same view number and sequence number as the PRE_PREPARE message, then it broadcasting the COMMIT message to all replicas including the primary.

A replica commits the corresponding REQUEST after it receives at least 2f matching COMMIT messages from other replicas. To prevent a faulty primary that intentionally delaying a message, the client starts a timer after it sends out the REQUEST message and waits for f+1 responses from different replicas. Assuming f replicas are faulty, at least one response must from a non-faulty replica. If the timer expires, the client broadcasts the REQUEST message to all replicas and suspects the primary. The rest replicas will then have an election to elect a new primary. In BFT algorithm, digital signature or authenticator is employed to ensure the integrity of the message, and a cryptographic hash function is used to compute message digests.

The normal case operation of BFT is illustrated in the Figure 2 below:

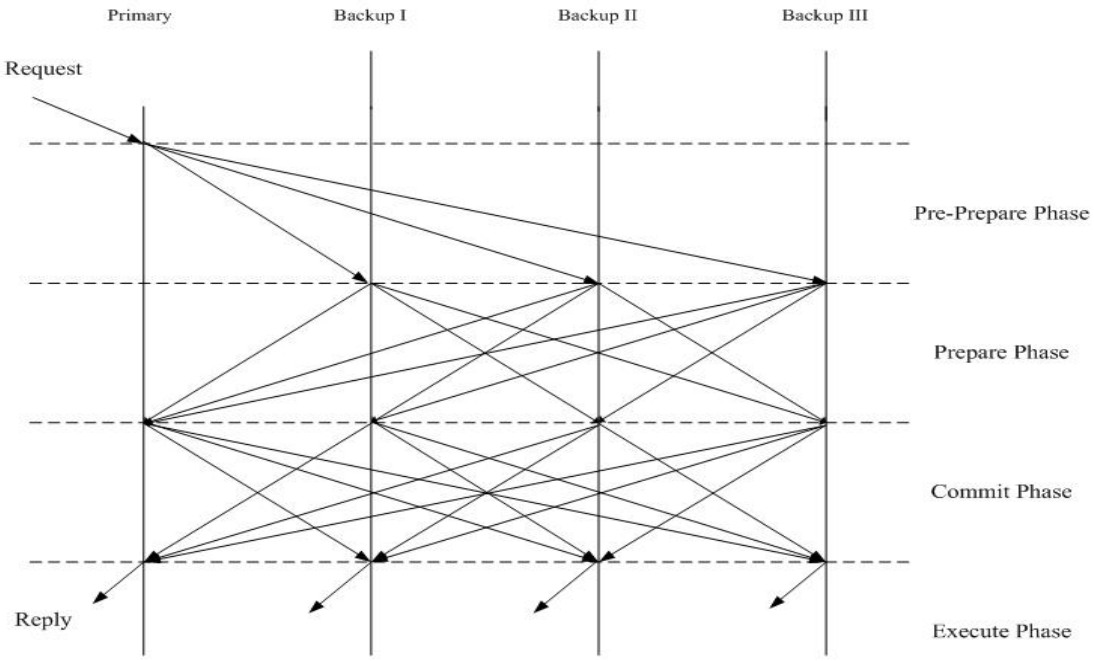

Figure 2: Normal Case Operation of BFT

2.3 Other Byzantine fault tolerant techniques

2.3.1 Paxos

Paxos[26] is a well-known fault-tolerant protocol that allows a set of distributed servers, exchanging messages via asynchronous communication, to totally order client requests in the benign-fault, crash-recovery model. One server is referred as leader who coordination the transaction. If the leader crashes or becomes unreachable, a view change occurs, allowing progress to resume in the new view under the reign of the new leader. Paxos requires at least 2f+1 server to tolerate f faulty servers. Only one reply is required to be delivered to the client due to the servers are not Byzantine.

In normal case operation, there is a single leader which is used to communicate with the rest of servers. Paxos uses two asynchronous communication rounds to globally order the client updates. The leader assign a sequence number to the client and proposes this assignment to the rest of servers in the first round. In the second round, any server agreed on the proposal will send an acknowledgment to the rest of servers. When a server receives acknowledgment from the majority of replicas, in other word, the majority servers have accepted the proposal – the server orders the corresponding update.

2.3.2 Threshold digital signatures

Another well-known fault tolerant protocol is threshold digital signature which often be referred as threshold cryptography that distributes trust among a group of participants to protect information (e.g. Threshold secret sharing [28]) or computation (e.g. Threshold digital signatures [29]).This mechanism is prompted by Fragmentation-Replication-Scattering (FRS) which initially designed to provide intrusion tolerance for file systems and was later ex-tended to object-based systems. A (k, n) threshold digital signature scheme allows a set of servers to generate a digital signature as a single logical despite (k-1) Byzantine faults. It divides a private key into n shares, each owned by a server, such that any set of k servers can pool their shares to generate a valid threshold signature on a message, *m*, while any set of less than k servers is unable to do so. Each server uses its key share to generate a partial signature on *m* and sends the partial signature to a combiner server, which combines

the partial signatures into a threshold signature on m. The threshold signature is verified using the public key corresponding to the divided private key.

RSA shoup scheme [29], a representative example of practical threshold signatures, allows participant to generate threshold signatures based on the standard RSA digital signature. It provides verifiable secret sharing (i.e., the ability to confirm that a signature share was generated using a valid private key share), which is critical in achieving robust signature generation in Byzantine environment.

CHAPTER III

BYZANTINE FAULT TOLERANCE FOR NONDETERMINISTIC APPLICATIONS

In this chapter, we first, describe the system model of ND-BFT, including its operation, communication model and cryptography techniques. Then, we present the threat analysis to show the importance of our protocol. After that, we provide a systematic classification of replica nondeterminism and illustrate each solution for different type of replica nondeterminism.

3.1 System Model

In this section, we present an overview of the system model which will be used in following chapters. This model defines the operations provided by the system,

assumptions on node failures and the communications infrastructure, and the cryptographic primitives available for use by the ND-BFT protocol.

3.1.1 Operations

ND-BFT provides support for the execution of general operations. These are distinct from simple reads and blind writes to services state, as provided by some previous protocols. Reads and writes only allow directly reading or overwriting objects at the server. General operations, however, allow for the execution of complex operations that may depend on current state at the server, and provide a far more power interface.

All operations should be deterministic, e.g., given a serialized order over a set of operations, each replica should obtain the same result in running each operation, provided they have the same application state, which is the purpose of this protocol.

3.1.2 Failure Model

Our system consists of a set $C= \{c_1 \ldots, c_n\}$ of client processes and a set $R= \{r_1,\ldots,r_{3f+1}\}$ of $3f+1$ server processes. Server processes are known as replicas throughout this thesis, as they replicate the server application for reliability.

Servers are categorized into either correct server or faulty server. A correct process is constrained to obey its specification, and follow the ND-BFT protocol precisely. Faulty processes may deviate arbitrarily from their specification: we assume a Byzantine failure model where nodes may adopt any malicious or arbitrary

behaviors. The difference between fail benignly (fail-stop) and those suffering from Byzantine fault is not described in this thesis.

The correct system operation is able to tolerate up to f simultaneously faulty replicas. Transient failures are considered to last until a replica is repaired and has reestablished a copy of the most recent system state. No guarantees are offered beyond failures, and the system may halt or return incorrect responses to client operations.

The number of faulty clients is not considered in this thesis. It is assumed that application-level access control is implemented to restrict clients write to modify only application state for which they have permission. A malicious client is able to execute arbitrary write operations on data it has permission to access, but cannot affect other application data nor put the system in an inconsistent state.

3.1.3 Communication Model

The communication model in this thesis is assumed as an asynchronous distributed system where nodes are connected by Ethernet. We place very weak safety assumptions on this network – it may fail to deliver messages, delay them, duplicate them, corrupt them, deliver them out of order, or forward the contents of messages to other entities. There are no bounds on message delays, or on the time to process and execute operations. We assume that the network is fully connected; given a node identifier, any node can attempt to contact the former directly by sending it a message.

For liveness, we require the use of fair links; if a client keeps retransmitting a request to a correct server, the reply to that request will eventually be received.

Liveness for the BFT module used by ND-BFT also requires the liveness conditions assumed by the BFT protocol. Notably, we assume that message delays do not increase exponentially for the lifetime of the system, ensuring that protocol timeouts are eventually higher than message delays. These assumptions above are not required for liveness that the message delay is not guaranteed based on those assumption.

3.1.4 Cryptography

Our protocol requires highly cryptography to ensure its correctness. Clients and replicas must be able to authenticate their communications to prevent forgeries. We assume that nodes can use unforgeable digital signatures to authenticate messages, using a public key signature schemes such as DSA. We assume a message *m* signed by node n as <*m*> and no node can send <*m*>, either directly or as part of another message, for any value of *m*, unless it is repeating a previous message or known n's private key. Any node can verify the integrity of a signature by the message *m* and n's public key.

We assume that the public keys for each node are known statically by all clients and replicas, or available through a trusted key distribution authority. Private keys must remain confidential, through the use of a secure cryptographic co-processor or otherwise. If the private key of a node is hacked, then the node is considered faulty.

The security of the communication between pairs of nodes, despite message transmission on untrusted links, is guaranteed by using Message Authentication Codes (MACs). Each pair of node shares a secret session key, established via key

exchange using public key cryptography. The notation $<m>u_{x,y}$ is used to describe a message authenticated using the symmetric key shared by nodes x and y.

A collision-resistant hash function is assumed in our protocol that that any node can compute a digest *hm* of message *m*, and it is impossible to find two distinct messages *m* and *m'* such that *hm=hm'*. The hash function is used to avoid sending full copies of data in messages for verification purposes, instead using the digest for verification.

Our cryptographic assumptions are probabilistic, but there exist signature schemes and hash functions for which they are believed to hold with very high probability. Therefore, we assume they hold with probability 1.0 in remainder of this thesis. To avoid replay attacks, we tag certain messages with nonce that are signed in replies.

3.2 Threat Analysis

This section explains the importance of replica consistency and the necessarily to import our protocol to tackle nondeterministic data.

Byzantine fault tolerance system, which based on state machine replications, must be deterministic to maintain the consistency of the system [12]. However, practical applications always contain some forms of nondeterminism. For example, the time-last-modified in a distributed file system is set by reading the server's local clock; if this were done in-dependently at each replica, the states of non-faulty replicas would diverge. When such applications are replicated to achieve fault and

intrusion tolerance, their nondeterministic behavior must be tackled to ensure the replicas consistency or totality.

The most difficult challenging for a software designer to designing a distributed application is the consistency of the disseminated information, and the control over the dissemination of that information. Therefore, the designer of a distributed system would wish for a transport layer that provides a guaranteed delivery-and-consistency of messages sent to multiple targets. Have such layer, most distributed applications become much easier to implement and maintain. Thus, the problem of consistency has received considerable attention when designing a distributed system. MIT-BFT framework [18] strongly relies on the total ordering of the message passed by each replica during its three phases. The total ordering of messages requires a consensus decision. Without the guarantee of the consistency of message in MIT-BFT framework, each replica might receives different request command at the same phrase that the system would have conflicting operations which may cause the crash of the entire system.

3.3 Type of Replica Nondeterminism

In the Byzantine fault tolerance algorithm [18] only one type of replica nondeterminism behavior has been recognized. In this section, we analysis different replica nondeterminism and classify them into three categories. Furthermore, we mainly focus on two types of replica nondeterminism and divide them into four types in order for us to build model to tackle their replica nondeterminism.

- Wrappable nondeterminism. A type of replica nondeterminism that can be simply controlled by an infrastructure-provided or application-provided wrapper function, without explicit inter-replica coordination. For instance, information such as hostnames, process ids, file descriptors, etc. can be determined group-wise. 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 [19].
- Per-determinable non-determinism. A type of replica nondeterminism whose value can be known before the execution of the request and it requires inter-replica coordination to ensure replica consistency.
- Post-determinable non-determinism. 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 completed until the end of the execution. It also requires inter-replica coordination to ensure replica consistency.

In this thesis, we merely focus on last two type of replicas nondeterminism since the wrappable replica non-determinism has been fully studied by [19] and can be tackled by wrapper function without inter replica coordination.

We further classify the replica nondeterminism into two following types based on whether a replica can verify the nondeterministic values proposed (or recorded) by another replica.

- Verifiable non-determinism. This type of replica nondeterminism whose values can be verified by other replicas.
- Non-verifiable non-determinism. This type of replica non-determinism whose values can not be fully verified by other replicas which means a replica might be able to partially verify some nondeterminism values proposed by another replica. This feature would help to reduce the impact of a faulty replica.

In order to implement current application or to develop new application to efficiently handle each type of replica nondeterminism, we classification gives fours types of replica nondeterminism of our interests:

- Verifiable pre-determinable non-determinism (VPRE). Previous study treated clock-related operations as this type of operation. However, strictly speaking, it is not possible for a replica to verify deterministically another replica's proposal for the current clock value without imposing stronger restriction on the synchrony of the distributed system (i.e., bounds on message propagation and request execution).
- Non-verifiable per-determinable non-determinism (NPRE). This type of non-determinism is exhibited as on-line gaming applications, such as Blackjack and Texas Hold'em. These application requires highly randomness to ensure the integrity of services [4], for instance, the card distributed to each player must be unpredictable. Such application depends on the use of good secure random number generators. For the security proposes, it is essential to make one's choice of a random number unpredictable, let alone verifiable by other replicas.

- Verifiable post-determinable non-determinism (VPOST). We have yet to identify a commonly used application that exhibits this type of non-determinism. We include this type for completeness.
- Non-verifiable post-determinable non-determinism (NPOST). This type of non-determinism is exhibited, in general, in all multi-threaded applications. Ideally, the replicas should collectively determine the set of nondeterministic values to prevent a single faulty replica from compromising the integrity of other replicas [10]. However, it is not clear if it is always feasible for replicas to apply a deterministic algorithm to decide on a common set of values from those reported by individual replicas, in case of multi-threading. Furthermore, it would require a test execution of every request at every replica, which might be too expensive to be practical. Therefore, our current solution is to reply on the information reported by a single replica (i.e., the primary replica) and to employ additional recovery mechanisms to minimize the impact of faulty of replica.

3.4 Solution for each type of replica non-determinism

In this section, we present the extensions of current BFT framework in handling all common types of replica nondeterminism. The unified framework requires closely coordination between BFT algorithm and the application be replicated. Comparing with the APIs used in BFT framework [18], the following server upcalls (i.e., callback functions registered by the server application) are modified:

Replica upcalls:

int propose_value(Seqno *seqno*, Byz_req **req*, int **ndet_type*, Byz_buffer **ndet*);

Here *seqno* is the sequence number assigned to the client's REQUEST, *req* is request message, *ndet_type* is the type of replica nondeterminism when executing client's REQUEST, and *ndet* is a pointer to the buffer that stores the nondeterministic values. This function returns appropriate values to indicate if the call successful. Both *ndet_type* and *ndet* are out-parameters, which mean the application is expected to set their values.

Check replica non-determinism:

int check_value(Seqno *seqno*, Byz_req **req*, int **ndet_type*, Byz_buffer **ndet*)

This function is used to check the type of replica nondeterminism, which is invoked when a replica want to verify the type of replica nondeterminism and the nondeterministic values received from the primary. The parameters in this function are the same as those in *propose_value*() function. The different between two function is the parameters *ndet_type* and *ndet* in this function are in-parameters, which means the information is passed to the application. The verification result is returned to the caller in the return value.

Replica execute:

int execute(Byz_req **req*, Byz_rep **rep*, Byz_vuffer **ndet*, int *cid*, bool *ro*)

In *execute()* function the signature is not modified, but the interpretation of one of its parameters is changed. Parameter *req* is REQUEST message, *rep* is REPLY

message to be generated by the replica, *ndet* is originally defined as a pointer to the nondeterministic values obtained from the primary replica and to be used by all replicas, i.e., it is intended to be used as in-parameter. It is not reinterpreted as an in-out parameter which is depending on the type of replica non-determinism, for instance, the parameter might be changed from in-parameter to out-parameter when a replica has post-determinable nondeterminism and the function is invoked at the primary replica.

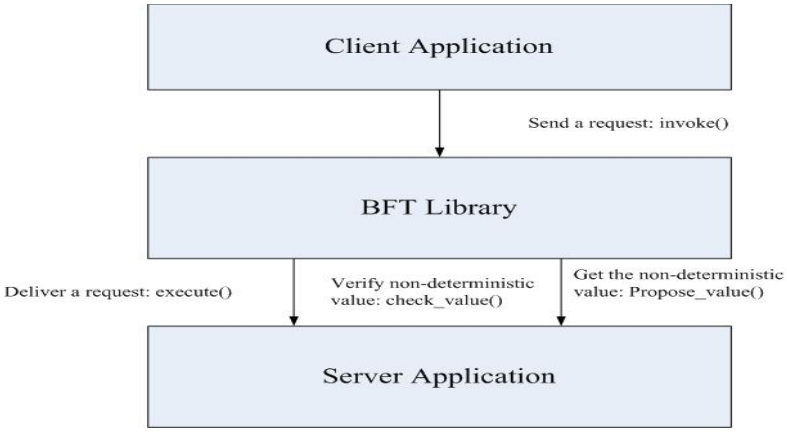

Figure 3: System Architecture

The replica nondeterminism we classified in previous section are defined in the form of four constant integer values as below:

- VERIFIABLE_PRE_DETERMINABLE
- NONVERIFIABLE_PRE_DETERMINABLE
- VERIFIABLE_POST_DETERMINABLE
- NONVERIFIABLE_POST_DETERMINABLE

The BFT algorithm is modified in following ways: when the client's REQUEST arrives at the primary, if it is ready to order the message (when the

number of ordered but not-yet executed message is smaller than the window threshold), the primary invokes the *propose_value()* callback function registered by the application layer. The application supplies the type of replica nondeterminism that would be involved in the execution of the request, and if applicable, the nondeterministic values. Depending on the type of replica nondeterminism returned by the application, the modified BFT algorithm operates differently according to the mechanisms described from section 3.4.1 through section 3.4.4.

We introduce two extra-phases: PRE-PREPARE-UPDATE, a phase before the PREPARE and POST-COMMIT phase, a phase after COMMIT phase into the new algorithm to handle replica nondeterminism in the modified BFT algorithm. We introduce two new types of control message, PRE_PREPARE_UPDATE message and POST_COMMIT message accordingly. The PRE_PREPARE_UPDATE message is used in PRE-PREPARE-UPDATE phase for the replicas to reach the Byzantine agreement on the collection of the nondeterministic values contributed by different replicas when non-verifiable pre-determinable non-determinism is present. The POST_COMMIT message is used in POST-COMMIT phase for the replicas to reach athe Byzantine agreement on the nondeterministic values recorded by the primary after it has executed a REQUEST message when post-determinable non-determinism is present.

3.4.1 Verifiable Pre-determinable Non-determinism(VPRE)

If the type of replica nondeterminism at primary is VPRE, the primary calls *propose_value()* function in its *ndet* parameter to propose the nondeterministic types

and values. Then it includes the nondeterministic information into the PRE_PREPARE message, and multicast the message to all replicas.

When the replica receives the PRE_PREPARE message, it calls *check_value()* function to pass the nondeterministic information to upper layer. Then it verifies the following information:

- The type of replica nondeterminism for the client's REQUEST is consistent with what is reported by the primary replica.
- The nondeterministic values proposed by the primary is consistent with its own values(not necessarily identical)

If the verification succeed, the replica will verify the nondeterminism type and value proposed by the primary. After that, it accepts the REQUEST and the ordering information, and it logs the PRE_PREPARE message and multi cast PREPARE message to all other replicas. The following steps work the same as the original BFT framework. On the other hand, if the verification fails, the replica will receive an error code returned by *check_value()* function. The replica will then suspect the primary. We illustrate the normal case operation in handling VPRE in Figure 4.

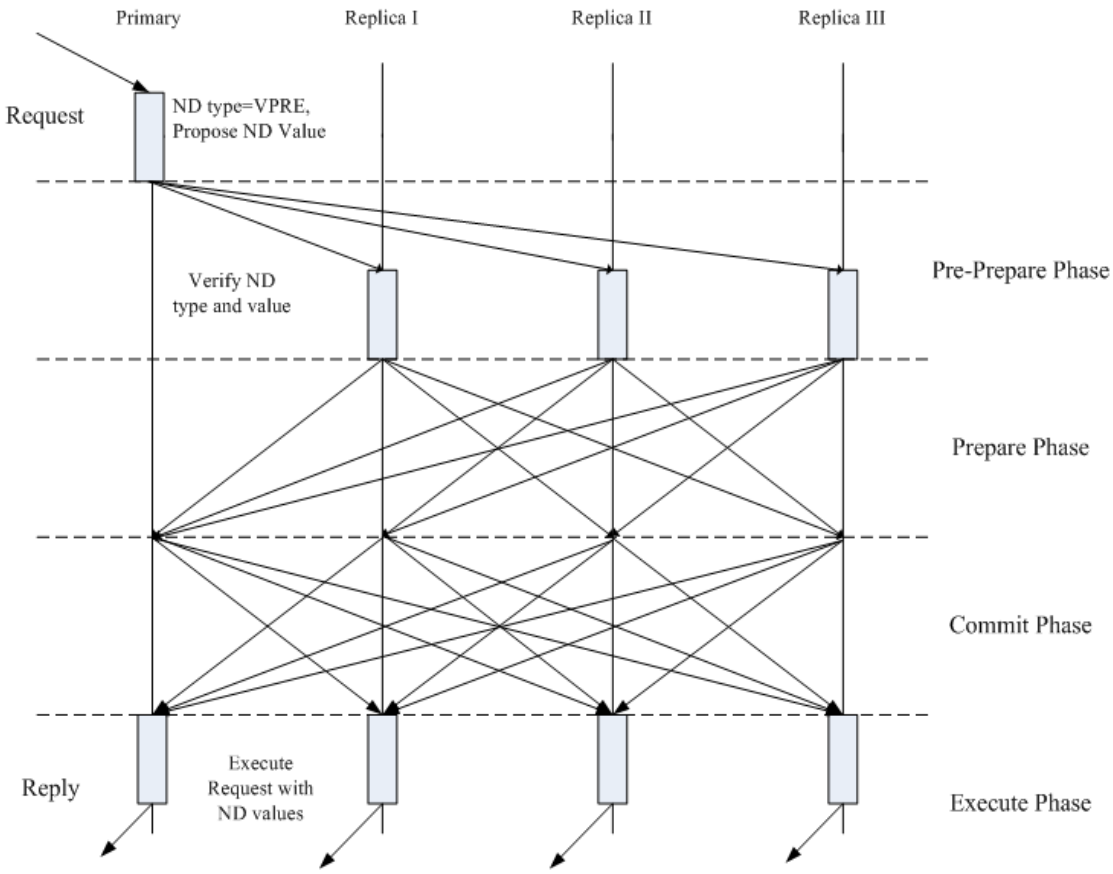

Figure 4: Solution to handle Verifiable Pre-Determinable Non-determinism

3.4.2 Non-Verifiable Pre-determinable Non-determinism(NPRE)

If the type of replica nondeterminism at primary is NPRE, that the replica cannot verify other replicas' nondeterministic value for this type of nondeterminism, consequently, the *propose_value()* function is called by the primary to propose its share of nondeterministic values in *ndet* parameter. The nondeterministic information is included in PRE_PREPARE message and the primary multicasts the message to all replicas.

When the replica receives the PRE_PREPARE message, it verifies the REQUEST message and ordering information from the primary. Since for this type of replica nondeterminism, the replica is not able to verify other replicas' nondeterministic value, the replica, for this type of replica nondeterminism, will only verify the nondeterministic type if the verification of REQUEST and ordering information is succeed. After the verification of the nondeterministic value in PRE_PREPARE message, the replica enters into PRE_PREPARE_UPDATE phase by building and sending the PRE_PREPARE_UPDATE message *<PRE_PREPARE_UPDATE, v, n, d, t, b>* to the primary, where *v* indicates the view number in which the message is being sent, *n* is the sequence number, *d* is the request message's digest, *t* is type of replica nondeterminism, and *b* is the value of replica nondeterminism.

After the primary collect at least 2f valid PRE_PREPARE_UPDATE message from different replica, it start to build PREPARE message, including 2f+1 (including the primary itself) sets of nondeterministic values, each message is protected by the proposer's digital signature or authenticator. The following steps operate according to the original BFT model, except that the PREPARE and COMMIT message also carry the digest of the nondeterministic values, and the 2f+1 set of nondeterministic values are delivered to the application layer as part of the *execute()* call. We illustrate the normal case operation in handling NPRE in Figure 5.

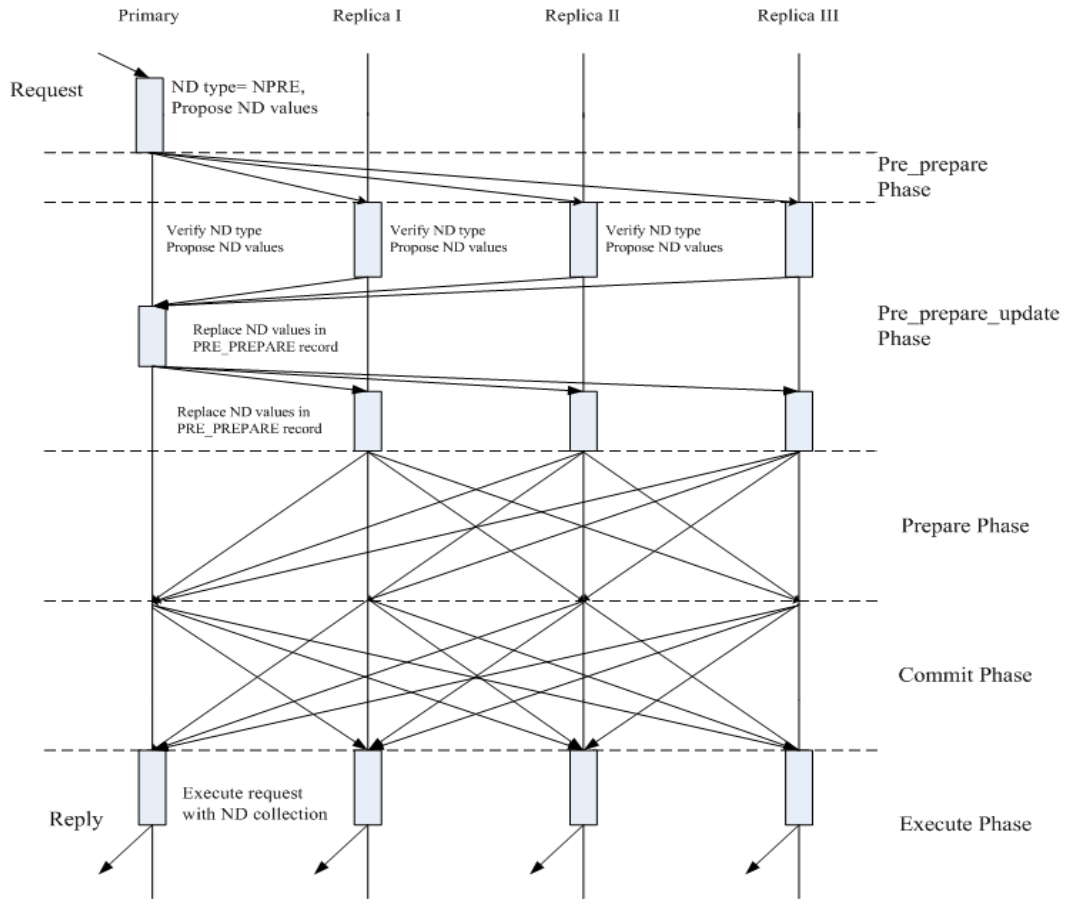

Figure 5: Solution to handle Non-Verifiable Pre-Determinable Nondeterminism

While we have described the mechanism to be used to handle this type of replica nondeterminism, it is necessarily for us to further discuss the type of applications that exhibit such replica nondeterminism and how our mechanism can be used to improve the security and dependability of such applications.

For applications such as online poker games [4], the source of replica nondeterminism is the most crucial state that should be protected since such values are used as the seeds for the pseudo-random number generator to generate a random number for the operations, such as shuffling cards. Such application replies on highly

randomness of their values to maintain the integrity of the system. The process of retrieving such nondeterministic values is often referred as entropy gathering (entropy is defined as a measurement of the randomness of the data). The value can be obtained either from hardware device, such as Geiger counter that counts the number of radioactive decays detected, or using software solution, such as through sampling keyboard or mouse events in a computer[14]. On the other hand, if such values are not obtained from a high-entropy source, they might be predictable since the pseudo random number generator is not truly random [14], and once seed is known, consequently the output data from random number generator would also be known. In practical, if the server of online poker game is compromised, and the seed which used to generate the random number, or in another word the seed used to shuffle the cards would be discovered by the person who hacked into the server. And he/she would be able to cheat in the game.

Here we assume that a faulty replica cannot transmit the confidential state to its colluding clients in real time. This can be achieved by using an application-level gateway, or a privacy firewall as described by Yin [3], to block illegal replies. A compromised replica may, however, replace a high entropy source to which it retrieves the nondeterministic values with a deterministic algorithm, and convey such algorithm via out-of-band covert channels to its colluding clients.

To counter such threats, such applications must be replicated using Byzantine fault tolerant algorithm. Furthermore, each replica uses different methodology to generate its nondeterministic values. In which case, a replica is in no position to verify the non-deterministic values proposed by another replica. Ideally, a replica should not

even know how other replicas generate their nondeterministic values, let alone to verify them.

For each operation that requires nondeterministic input, the replicas should collectively determine the input by applying the mechanism described in this section which is essential in the entire operation, because otherwise, a single replica might be able to compromise the whole service (despite the fact that there are at least 3f+1 replicas employed), which would jeopardize the intent of applying Byzantine fault tolerance to such applications.

3.4.3 Verifiable Post-determinable Non-determinism(VPOST)

If the type of replica nondeterminism at primary is VPOST that the nondeterministic value cannot be known before the execution of the request, the primary, under this circumstance, only includes the nondeterministic type in the PRE_PREPARE message without enclose any nondeterministic values. Then, the primary multicasts the message to all replicas.

When the replica receives the PRE_PREPARE message, it verifies the REQUEST message and the ordering information. If the verification succeed, the replica will confirm the nondeterministic type associated with the REQUEST message. The protocol then proceed to the COMMIT phase as usual. Otherwise, the replica suspect the primary.

On receiving the returned parameters, it enters POST-COMMIT phase by building POST_COMMIT message< <*POST_COMMIT, v, n, d, t, b*>,*m*>, where *m* is the REQUEST message from client, *b* is the post-determined non-deterministic

values, *d* is the digest of the REPLY. The primary, first, stores the in-formation into the *postnd* log, and then it multicast the message to all replicas and sends the REPLY message back to the client.

The replica will deliver REQUEST message if the Byzantine agreement on the nondeterministic values for the REQUEST has been reached. If fail to reach the agreement, or the verification of nondeterministic value is incorrect, the replica will suspect the primary. Furthermore, the replica will suspect the primary if the REPLY does not match with the REPLY's digest from the primary. However, despite the result of the comparison, the replica produces the same REPLY using the same set of nondeterministic values. The detailed processes describe as follow: when the replica receives the POST_ COMMIT message from the primary, it checks the received nondeterministic values through the *check_value()* upcall. If the verification succeed, the replica re-multicasts the POST_COMMIT message with its own signature or authenticator to the rest of the replicas. Otherwise, the replica suspects the primary. When a replica receives at least 2f POST_COMMIT messages, which its nondeterministic values matches with other replicas', it delivers the REQUEST message through the *execute()* upcall together with the verified non-deterministic values. The replica then sends the REPLY to the client while the *execute()* call returns.

A POST-COMMIT phase is required for the primary to disseminate the information in the *postnd* log to duplicate the information and for all correct replicas to ensure that they have received the same set of values for the corresponding REQUEST. Unlike the PRE-PREPARE-UPDATE phase for controlling NPRE, the POST-COMMIT phase involves with the entire steps needed for correct replicas to

reach the Byzantine agreement on the nondeterministic values. It requires three rounds of message exchange similar to those used to determine the ordering of the requests under normal case operations. For NPRE, the PREPARE and COMMIT phase are needed for the correct replicas to reach byzantine agreement on the nondeterministic values. The nondeterministic values are integrated into the corresponding request message. Due to the ordering information for the corresponding request has already been decided, we could not do so for post-determinable nondeterminism. We illustrate the normal case operation in handling VPOST in Figure 6.

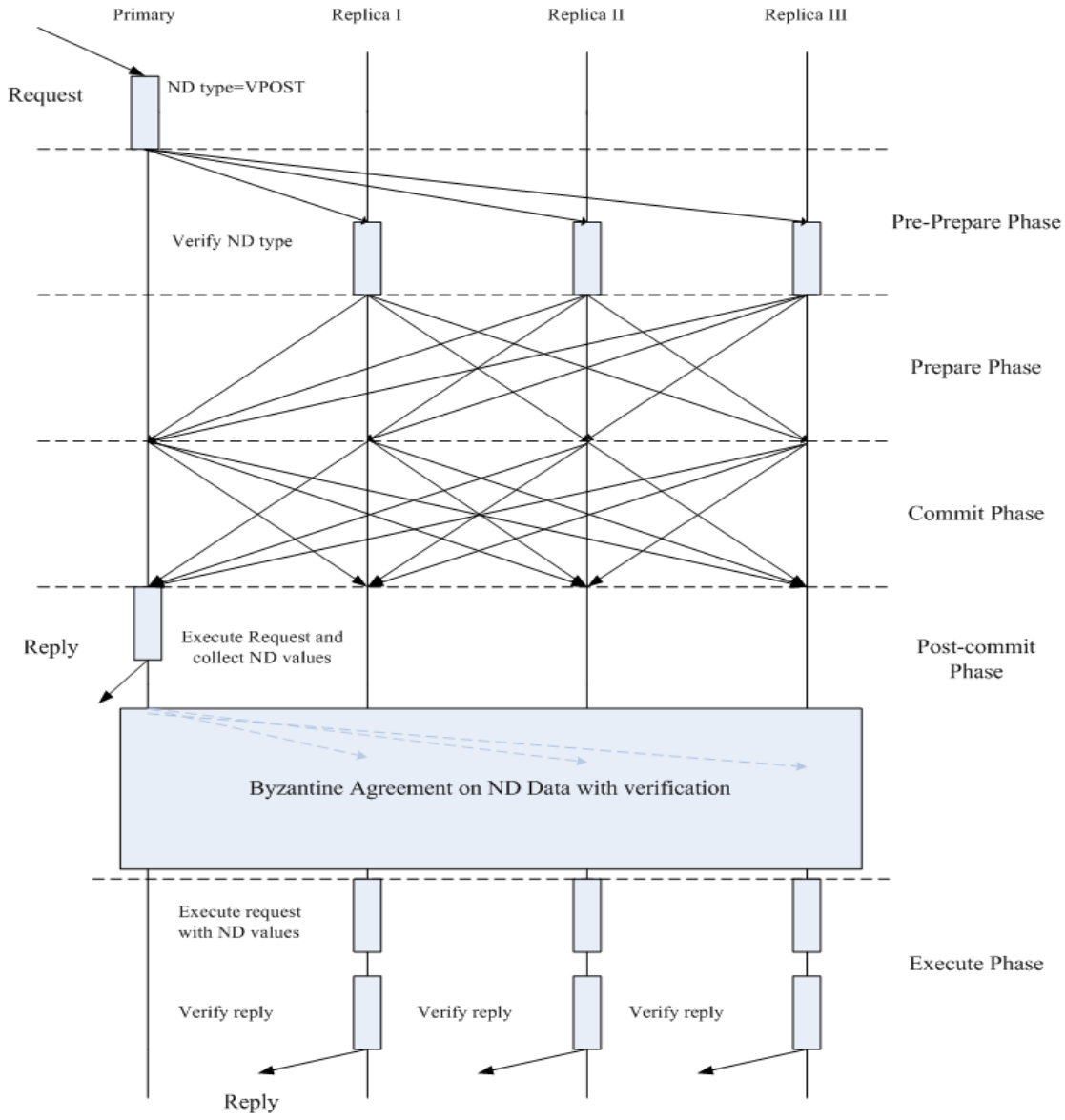

Figure 6: Solution to handle Verifiable Post-determinable Nondeterminism

3.4.4 Non-Verifiable Post-determinable Non-determinism(NPOST)

If the type of replica nondeterminism at the primary is NPOST, the way to handle such replica nondeterminism involves with the similar step as the way to

handle VPOST as those described in previous section until the replica deliver the REQUEST with post-determined nondeterministic values, as shown in Figure 6.

When the primary invokes the *execute()* upcalls and receives the REPLY and non-deterministic values. It enters the POST-COMMIT phase by sending the REPLY to the client. And then, it builds and multicast a POST_COMMIT message with following information:

- The identity information for the REQUEST message such as the sequence number assigned to the message, the view number, and the digest of the message.
- The recorded nondeterministic values.
- The digest of the REPLY message.

When replica receives the POST_COMMIT messages, it verifies the REQUEST information and re-multicast the message with its own signature or authenticator to all replicas. Until the replica has collected at least $2f$ POST_COMMIT messages which match with nondeterministic values from other replicas, it prepares for the execution of the REQUEST message.

We must realize that a malicious primary may cause the confusion of the replicas or block them from providing useful services to corresponding clients by disseminating a wrong set of nondeterministic values. For instance, if the nondeterministic data contains thread ordering information, a malicious primary can arrange the ordering in such a way that it may lead to the crash of the replicas (e.g., if the primary knows the existence of a software bug that leads to a segmentation fault), or it may cause a deadlock at the replica (it is possible for a replica to perform a

deadlock analysis before it follows the primary's ordering to prevent this from happening).

Since in general the replica cannot completely verify the correctness of the nondeterministic values until it actually executes the request, it is important for a replica to launch a separate monitoring process before invoking the *execute()* call. If the replica runs into a deadlock or a crash failure, the monitoring process can restart the replica and suspect the primary.

If the replica can successfully complete the *execute()* upcall, it compares the digest of its own REPLY message with that received from the primary. If those two do not match, the replica suspects the primary. Regardless of the comparison result, the replica sends the REPLY message to the client. It is safe to do so because if all correct replicas produce the similar REPLY using the same set of nondeterministic values(even if they might be different with the set actually used by the primary replica, which implies that the primary is lying and suspicious), the result is valid.

A good example of this type of replica non-determinism is that of multi-threaded applications [13]. When such applications are replicated, we must ensure different threads access the shared data in the same order, otherwise, the state of different replicas may diverge. Due to the complexity and dynamic nature of multi-threaded applications, it is virtually impossible to pre-impose an access ordering before the execution of a REQUEST. The only practical solution appears to be executing a REQUEST at one replica, recording the access ordering of threads on shared data, and propagating the ordering information to other replicas so that they

follow the same thread ordering, as described above. We illustrate the normal case operation in handling NPOST in Figure 7.

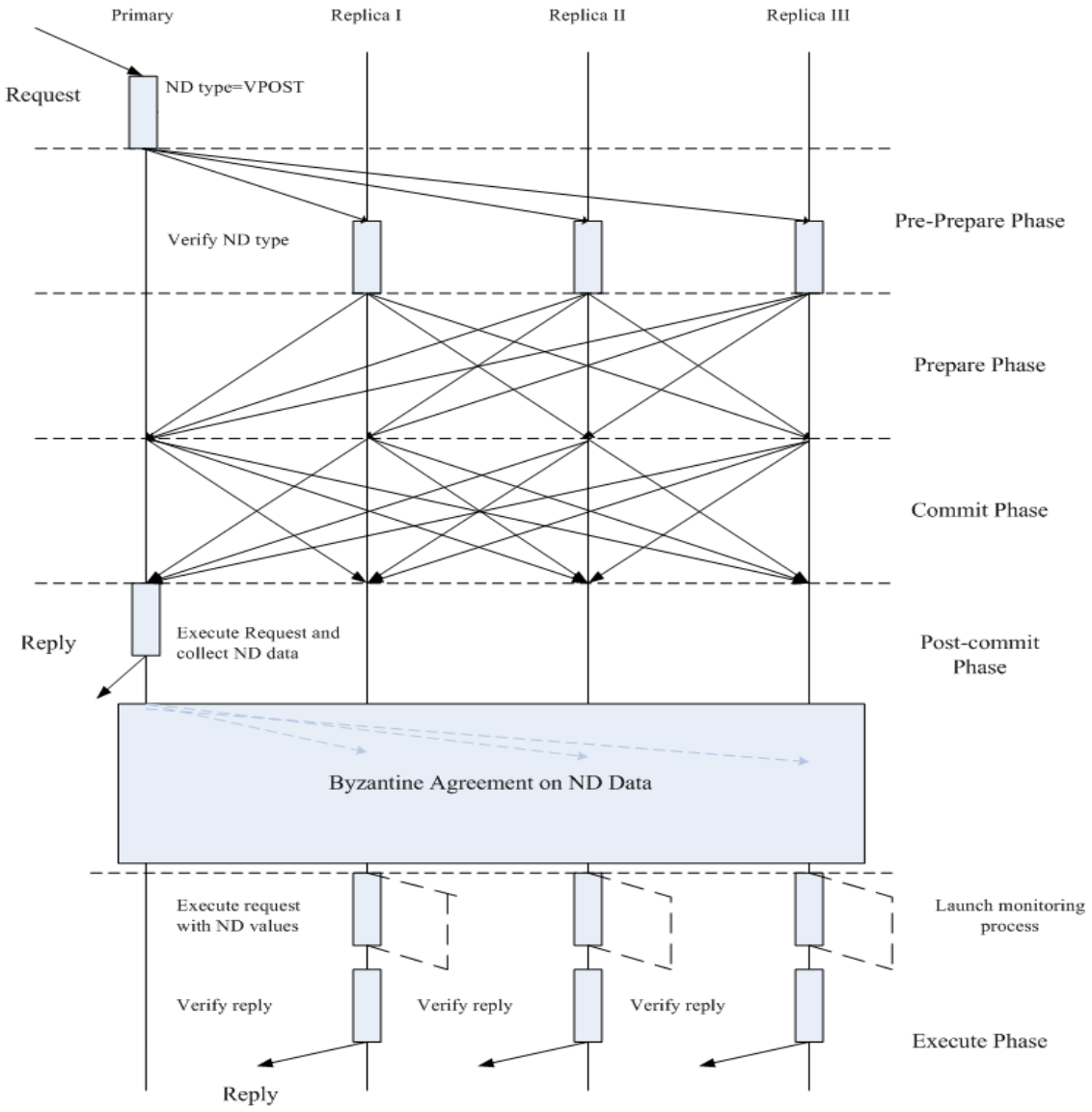

Figure 7: Solution to handle Non-Verifiable Post-determinable Non-determinism

3.5 Proof of Correctness

In this section we provide a proof of correctness of our mechanisms.

> Theorem 1: If a correct replica delivers a REQUEST *m* with a set of nondeterministic data in view *v*, then no other correct replica delivers *m* with a different set of nondeterministic data, and all such correct replicas use, or record (at the primary), the same set of nondeterministic data during its execution for *m*.

For VPRE, the primary replica proposes the nondeterministic data which combine with the agreement on it is carried out together with the REQUEST. At the end of the three-phase BFT algorithm, if some correct replicas agree on the ordering of the REQUEST, they reach an agreement on the nondeterministic data as well. For NPRE, the nondeterministic information is determined by the PRE-PREPARE-UPDATE phase, and it is followed by three phase BFT algorithm. The correct replica commits both the REQUEST m itself and reach the agreement on the associated nondeterministic data. For both VPRE and NPRE, when the REQUEST m is delivered at a correct replica, the non-deterministic data have been agree-upon are also delivered and used for execution.

For VPOST and NPOST, the three-phase BFT algorithm agrees on the non-deterministic data among correct replicas during the POST-COMMIT phase. When a correct replica receives the REQUEST *m*, it also receives the nondeterministic data accompanied with *m*. A correct primary must log the nondeterministic data during the execution of *m*, and have disseminated the data to replicas during POST-COMMIT phase. Therefore, the same nondeterministic data are used for execution at the correct client and other correct replicas.

CHAPTER IV

IMPLEMENTATION AND PERFORMANCE EVALUATION

4.1 Implementation

Our Byzantine fault tolerance for nondeterministicmapplication framework is built by implementing MIT-BFT framework. Th open-source library from MIT. We referred our implemented library as ND-BFT. And ourThe framework itself is composed as a generic prog library with a simply interface. Section 4.1 describes the library's implementation implementation of ND-BFTpresents its interface. To test our ND-BFT library in real world application and for future research purposes, we developed a poker game and used our imphe poker game with our ND-BFT library, which described in section 4.3.

4.1.1 Library

ND-BFT library uses a connection model of communication. The communication among each node is implemented using TCP, and multicast to the group of replicas is implemented using TCP over IP multicast. The IP multicast group contains all replicas while clients are not members of the multicast group. Replicas and clients are structured as a set of handlers that containing a handler for each message type and a handler for each timer. The handling loop works as following: Replicas and clients wait in a select call for a message to arrive or for a timer deadline to be reached and then they call the appropriate handler. The handler performs computations similar to the correspond action in the formalization, and then it invokes any methods corresponding to internal actions whose pre-conditions become true.

The SFS cryptography library is used to implement the public-key crypto-system with a 1024-bit modulus to establish 128-bit session keys. All messages are authenticated using message authentication codes computed using these keys and UMAC32. Message digests are computed using MD5.

For our new protocol, the public-key cryptography encryption and decryption are implemented to sign and verify the PRE_PREPARE_UPDATE and POST_COMMIT messages. These signatures are non-existentially forgeable even with an adaptive chosen message attack. MD5 still provide adequate security and it can be replaced easily by more secure hash function at the expense of some performance degradation.

In previous section we described our protocol messages at a logical level without specifying the size and layout of the different fields. While it is premature to specify the detailed format of protocol messages without further experimentation, but to understand the performance results in the next two chapters, it is important to describe the format of PRE-PREPARE-UPDATE and POST-COMMIT, we also describe the format of REQUEST and REPLY message in Figure 8 for the better understand of the normal case operation.

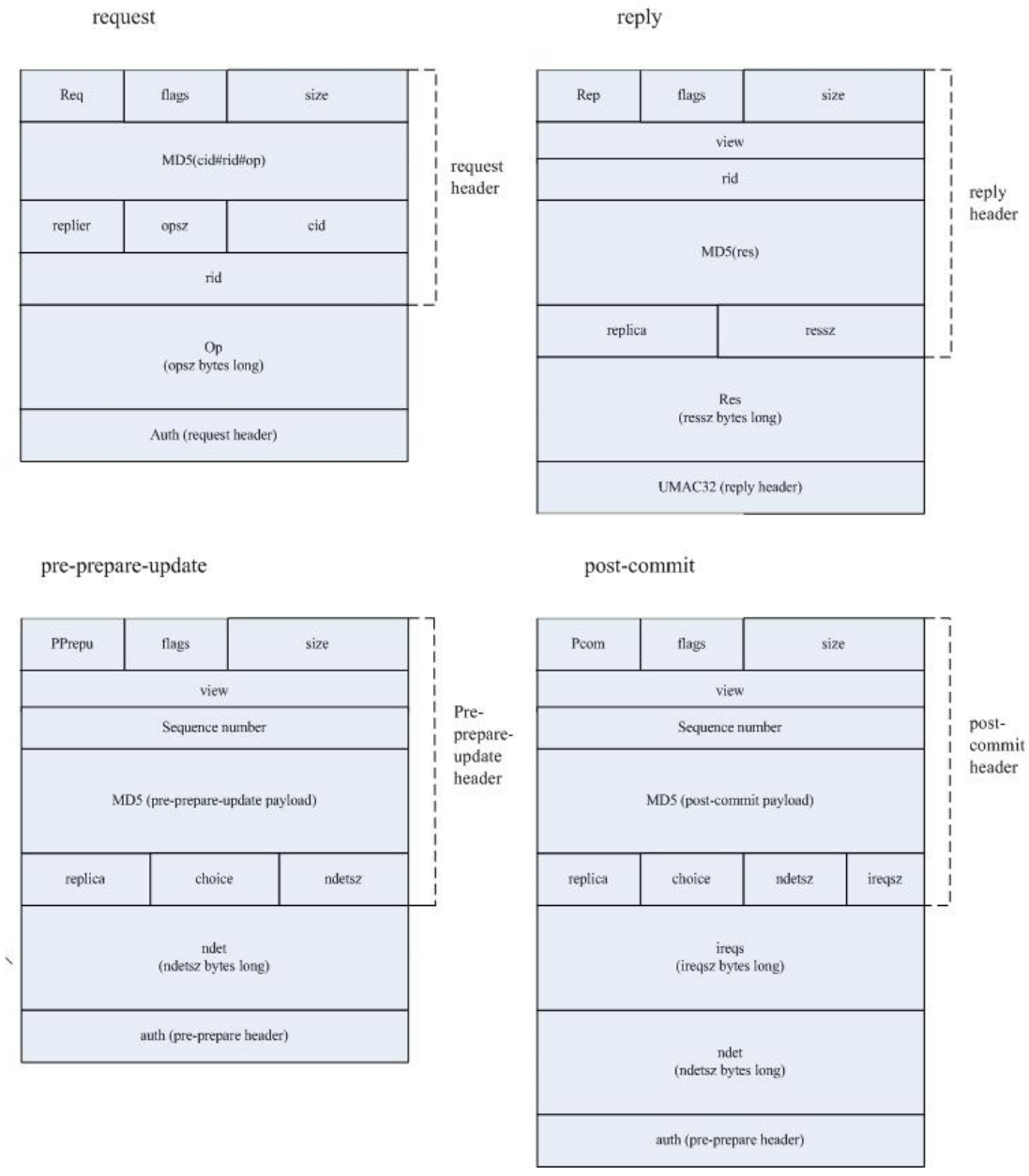

Figure 8: Message Format

The REQUEST header includes a MD5 digest of the string obtained by combined by the client identifier, *cid*, the REQUEST identifier, *rid*, and the operation being requested, *op*. It also includes the identifier of the designated replier. The flags

in the REQUEST header indicates whether to use the read-only optimization and whether the REQUEST contains a signature or an authenticator. In the normal case, all requests contain authenticators. In addition to the header, the REQUEST message includes a variable size payload and an authenticator. In the normal case, all REQUEST messages contain authenticators. The authenticator is composed of a 64-bit nonce, and n 64-bit UMAC32 tags that authenticate the REQUEST header. When a replica receives a REQUEST, it checks if the corresponding MAC in the authenticator and the digest in the header are correct.

The PRE_PREPARE_UPDATE message is assigned by the replicas when encounter VPRE. The PRE_PREPARE_UPDATE header is composed of a view number *v*, a sequence number *n* and an MD5 digest *d* of the PRE_PREPARE_UPDATE payload, the REQUEST message's id, a buffer that can be filled with nondeterministic choice, and a number of bytes in the nondeterministic values associated with the batch. The following payload includes the type of replica nondeterminism. Additionally, the message includes an authenticator with a nonce, and n-1 UMAC32 tags that authenticate the PRE_PREPARE_UPDATE header.

The POST_COMMIT message is used to handle VPOST and NPOST. The POST_COMMIT header includes the view number *v*, the sequence number *n*, MD5 digest *d* of the POST_COMMIT payload, the replica's id, choice, *ndetsz*, and the number of bytes in request inlined in the message, *ireqsz*. The variable size payload includes the requests that are inlined, *ireqs*, and the nondeterministic choices, *ndet*. The message also includes a corresponding authenticator.

After the replica executes all the operations in the batch, it sends a reply to the client. The reply header includes the view number *v*, the request identifier, *rid*, and MD5 digest *d* of the operation result, the identifier of the replica, and the size of the result in bytes, *ressz*. Additionally, the reply message contains the operation result if the replica is the designated replier. The other replicas omit the result from the REPLY message and set the result size in the header to -1. REPLY message contains a single UMAC32 nonce and a tag that authenticates the REPLY header. The client checks the MAC in the REPLY it receives. Client also checks the result digest in the REPLY with the result.

4.1.2 Interface

The algorithm is implemented as a library with a very simple interface which invokes some part of the library on client and some part on replicas.

On the client side, an initialization procedure is provided by library for the client using a configuration file, which contains the public keys, the IP address, and the port number of the replicas. The library also provides a procedure, *invoke()*, and which is called to execute an operation. The procedure is responsible for the protocol in the client side and returns the result when enough replicas have responded. The library also provides a split interface with separate send and receives calls to invoke requests.

On the server side, we provide an initialization procedure that takes an argument: a configuration file with the public keys and IP addresses of replicas and clients, the region of memory where the service state is stored, a procedure to execute

requests, and a procedure to compute nondeterministic choices. When our system needs to execute an operation, it does an upcall to the execute procedure. The argument to this procedure includes a buffer with the requested operation and its arguments, *req*, and a buffer to fill with the operation result, *rep*. The execute procedure execute the operation for the service, using the service state. As the service performs the operation, each time it is about to modify the service state, it calls the modify procedure to inform the library of the locations about to be modified. When the primary receives a request, it selects a non-deterministic value for the request by making an upcall to the *nondet* procedure. The nondeterministic choice associated with a REQUEST is also passed as an argument to the execute upcall.

4.1.3 Online Poker Game

We implement one online poker game, very familiar as Texas Holdem poker game, a client/server based web application which supports multi-player network players, and the ND-BFT library is installed on the server side. The type of replica nondeterminism for this application is VPRE Figure 9 shows the architecture of this game, as we have described in previous chapter. Because the purpose of creating this game is merely to test the performance of our library that running under a practical system, this game do not have complex GUI or structure that would slow down the system performance. Figure 9 shows the architecture of this game.

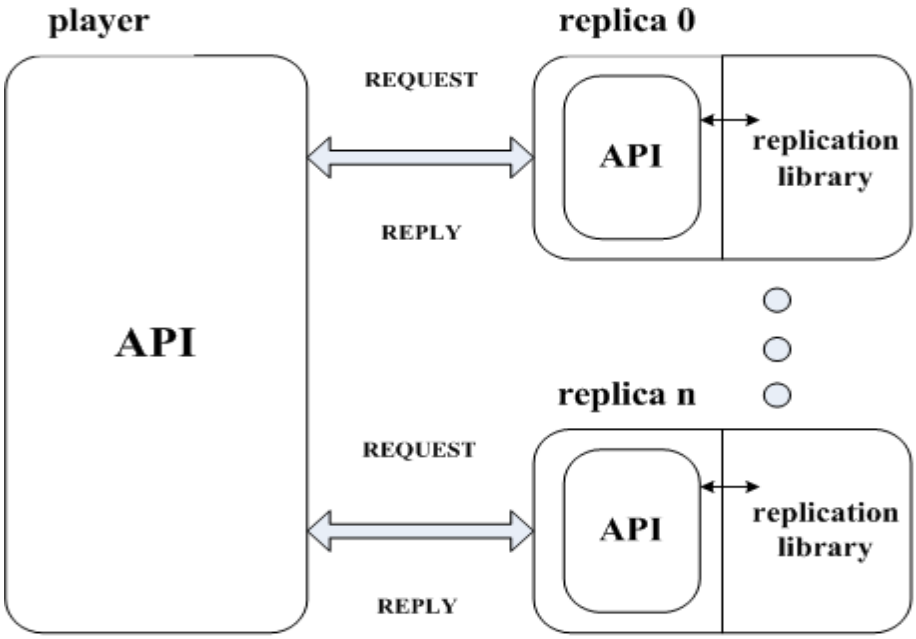

Figure 9: Architecture of online poker game with ND-BFT library

The normal operation of our game runs as the following:

On the client side, the client, first, establish a connection to the servers. And then, according to the pre-configured configuration file, which defines the number of player. For instance, if the number is 4, so if four clients connect to the servers, then the game starts. Each player in the game sends the request to invoke the *shuffling* function in server, and waiting for the reply from server. The player will pick the majority reply from servers to make a final decision.

On the server side, each server initiate according to the configuration file which also containing the IP address and port number, and then it waiting for enough player to join the game. On noticed there are enough players, the server piggybacks the acknowledgment information to client and waiting for client's request. On

receiving the request, the server invokes the *shuffling* function which triggers the ND-BFT library to handle the nondeterministic values. The execution of the clients' request will be used to seed the random number generator to generate a random number, and the output of the random number will modules by 52 to have a corresponded number as a poker card to the player.

4.2 Performance Evaluation

The BFT library can be used to implement Byzantine-fault-tolerant systems but these systems will not used in practice unless they perform well. This section presents results of experiments to evaluate the performance of these systems. These results show that these two extra phases we introduced in order to handle replica nondeterminism under different circumstances do not degrade performance significantly.

The experiments were performed using the setup in section 4.2.1. We describe experiments to measure the value. Section 4.2.2 use benchmarks to evaluate the performance during the normal case without checkpoint management, view changes or recovery.

We implemented the core mechanisms in C++ and integrated them into the BFT framework. The experiments described below are focused on the evaluation of the cost for providing Byzantine fault tolerance to nondeterministic applications in the BFT layer. The cost associated with recording nondeterministic values, verifying such values, and replaying such values in the application layer is not studied in this work.

4.2.1 Experimental Setup

The experiment consists of 14 nodes running RedHat 8.0 Linux. Of the 14 computers, 4 of them are equipped with Pentium-4 2.8GHz processors and the rest of those computers have Pentium-3 1GHz processors. The computers are connected via a 16-port Netgear 100Mbps switch. The replicas run on Pentium-4 nodes and clients are distributed across the rest of nodes.

4.2.2 Normal Case Operation

The experiment involves end-to-end latency and throughput measurements for client-server application under normal operations for different types of replica non-determinism, including composite types. Because of the experiments limitation, we only enable 4 replicas to take care a single Byzantine fault. The rest of the servers act as clients, and one server can be used as several clients with different port number. In each iteration, each client issues a request to the server replicas and waits for the corresponding reply. There is no waiting time between consecutive iterations. The size of each request and reply are kept fixed at 1KB. In each run, we measure the total elapsed time for 10,000 consecutive iterations at each client. From the measured time, we derive the average end-to-end latency for each of the request-reply iteration and the system throughput.

Figure 10 and 11 shows the end-to-end latency performance testing of our library under the normal case operation with different type of replica nondeterminism. Figure 10 shows the result of single type of replica nondeterminism which means the replica only containing one type of replica nondeterminism including VPRE, VPOST,

NPRE and NPOST. Figure 11 shows the result of composition type of replica nondeterminism that the replica containing two types of replica nondeterminism. With the increasing complexity construction of real-world applications, they could have more than one type of replica nondeterminism. In our experiment we only consider the applications involving two type of replica nondeterminism. The composited type of replica nondeterminism in our experiment includes VPRE+NPRE, VPRE+VPOST, VPRE+NPOST, NPRE+VPOST, NPRE+NPOST and VPOST+NPOST.

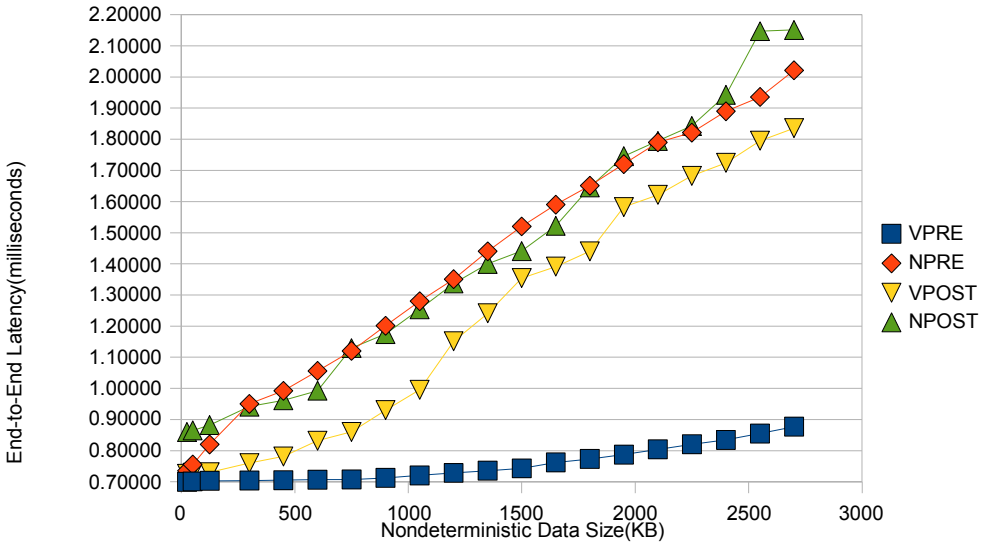

Figure 10: End-to-End Latency of Pure Nondeterminism

The type of replica nondeterminism and the size of nondeterministic values vary in different experiments, except for the throughput measurements, where the non-deterministic values are kept at 256 Bytes for each type. Note that log the nondeterministic values shown in the horizontal axis in Figure are for each type. That means, for composite types, the total size of nondeterministic value is twice times as large as those displayed.

Obviously, we can see from the previous figures that the latency of VPRE non-deterministic operation is noticeably smaller than that of other three nondeterministic operations. That is because except for VPRE, the handling of other types of non-determinism involves with one more phases of message exchanges for correct replicas to reach an agreement on the nondeterministic values. As such, as shown in Figure 9, the end-to-end latency is noticeably larger, and the throughput is smaller, compared with that of VPRE nondeterministic operations. The end-to-end latency difference is more significant as the size of nondeterministic values involved with each operation increases. Since our system deploys a lightweight fault-tolerant protocol, we expect it to achieve performance comparable to existing byzantine fault-tolerant replication protocol. We compare the throughput performance of original protocol where the replicas are deterministic with replica with different type of nondeterministic value. From the comparison, we can see that the throughput for deterministic replica is slightly higher than our system that handling different type of replica nondeterminism, which is acceptable due to the complexity of our mechanisms.

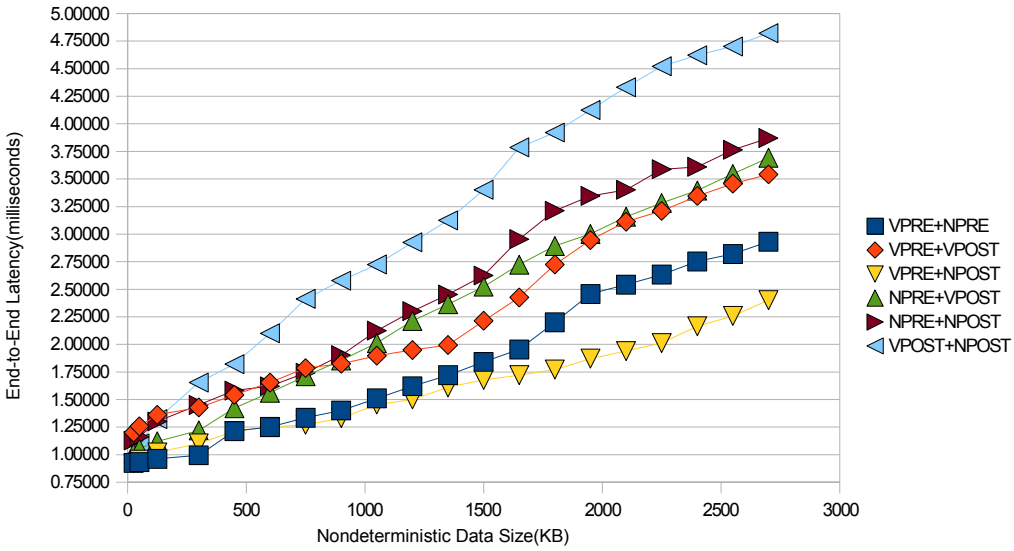

Figure 11: End-to-End Latency of Composite Nondeterminism

The results shown in Figure 10 and Figure 11 are obtained after a number of optimizations to the mechanisms described previously. Without these optimizations, the latency is significantly larger and the throughput is much lower, except for those from VPRE nondeterministic operations.

In the PRE-PREPARE-UPDATE phase, which is needed to handle NPRE and other composite types involving with NPRE, each replica multicasts its contribution of the nondeterministic values to all other replicas, and the primary decides on the collection (must include the contributions from 2f+1 replicas, including its own) to be used to calculate the final nondeterministic values. Instead of multicasting the collection of nondeterministic values, the primary disseminates the collection of the digests of the values proposed by each replica. This sharply reduces the message size if the size of nondeterministic values is large. Since each replica can log the

nondeterministic values received from other replicas, a replica can verify the digests provided by the primary replica using its local copies. A replica might not have received the values proposed by one or more replicas included in the primary's message, in which case, the replica asks for retransmission of the values.

During the POST-COMMIT phase, which is needed to handle NPOST non-determinism, the data in the *postn* log is piggybacked with the PRE_PREPARE message for the next REQUEST. This way, the Byzantine agreement for the nondeterministic values is reached together with that for the ordering of that REQUEST, which reduces the number of messages needed to handle this type of replica nondeterminism. Even though the end-to-end latency for a REQUEST increases slightly as a restem throughput is significantly improved. To avoid waiting indefinitely for the next REQUEST, the primary sets a timer. When the timer expires, the primary initiates the Byzantine agreement phases for the nondeterministic values in conjunction with a null REQUEST so that the existing mechanisms can be reused.

It may be surprising to see that the end-to-end latency for a REQUEST with NPRE is similar to, or slightly larger than, that for a request with NPOST when there are large quantity of nondeterministic values. With the above optimization, the PRE-PREPARE-UPDATE phase involves with at least two large messages (one message per replica on its proposed nondeterministic values) while the POST-COMMIT phase (needed to handle NPOST) involves with only one large message (sent by the primary). Due to the same reason, the throughput for requests with NPOST is higher for those with NPRE when enough concurrent clients are present (so that virtually all

post-determinable nondeterministic values are piggybacked with the PRE_PREPARE messages for other requests, rather than being sent as separate messages).

Figure 12 shows the result of throughput performance for pure replica non-determinism. And accordingly Figure 13 shows the throughput for composite replica nondeterminism.

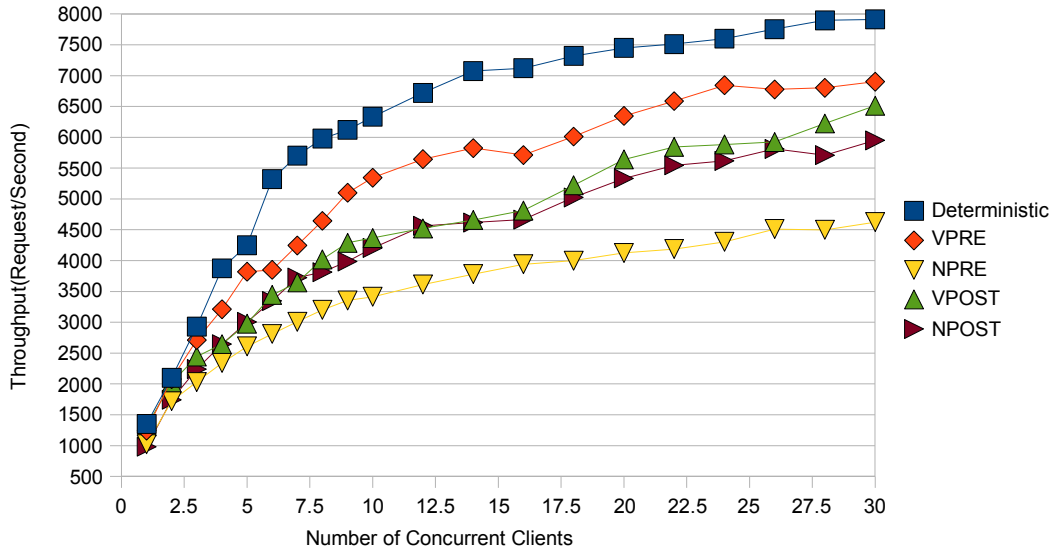

Figure 12: Throughput of Pure Nondeterminism

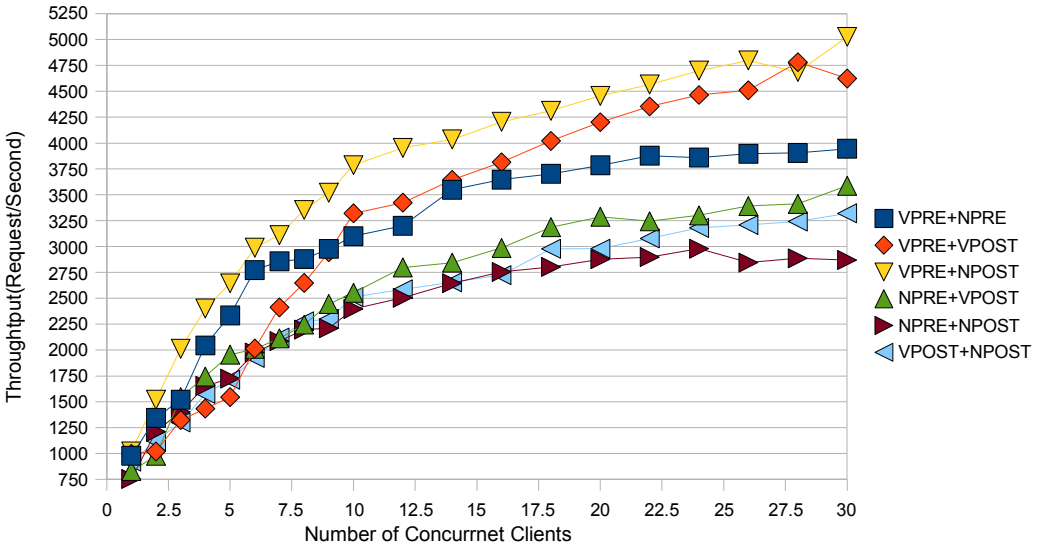

Figure 13: Throughput of Composite Nondeterminism

4.2.3 Online Poker Game

To demonstrate ND-BFT's performance on real application, we conducted the experiments for our online poker game replicated with ND-BFT library. The programming language we used to develop the online poker game is Java. We use JNI (Java Native Interface) technique to connect the online poker game with ND-BFT library. The experiments include throughput measurement with different number of players. We run the experiments using the same network environment as the experiment for ND-BFT library. For players who request to the replica will only issue one request to the replica to invoke the *shuffling* function to shuffle the card. On receiving the command from the player, each replica runs 1,000 consecutive iterations for the card shuffling. There are no waiting times for the players. We measure the system throughput by calculating the elapse time. We analyze the performance of online poker game without view-changes or proactive recovery. We start by

presenting results of experiments that ran with four replicas. We conduct the second experiment with seven replicas (may tolerate two faulty replicas).

For comparison purposes, the size of each request and reply still kept at 1KB. As we described in previous section, online poker games require a seed to generate a random number which always containing NPRE. To exhibit our algorithm could be applied on practical applications, we compare the performance between the poker game with and without our library; we only wrote code for it to work in the normal case.

Figure 14 present results of the throughput performance comparison between the original online poker game and the replicated online poker game, respectively, in a configuration with four replicas. The comparison between ND-BFT and NO-REP shows that if there are less than four players, the performance of ND-BFT is close to the performance of NO-REP. The throughput of ND-BFT increase rapidly when there are more than four players in the game. Percentage-wise, the comparison of the throughput performance is lowered by 30% to nearly 50%, which indicated that this library would be more efficient when running under lightweight environment which have a small number of players.

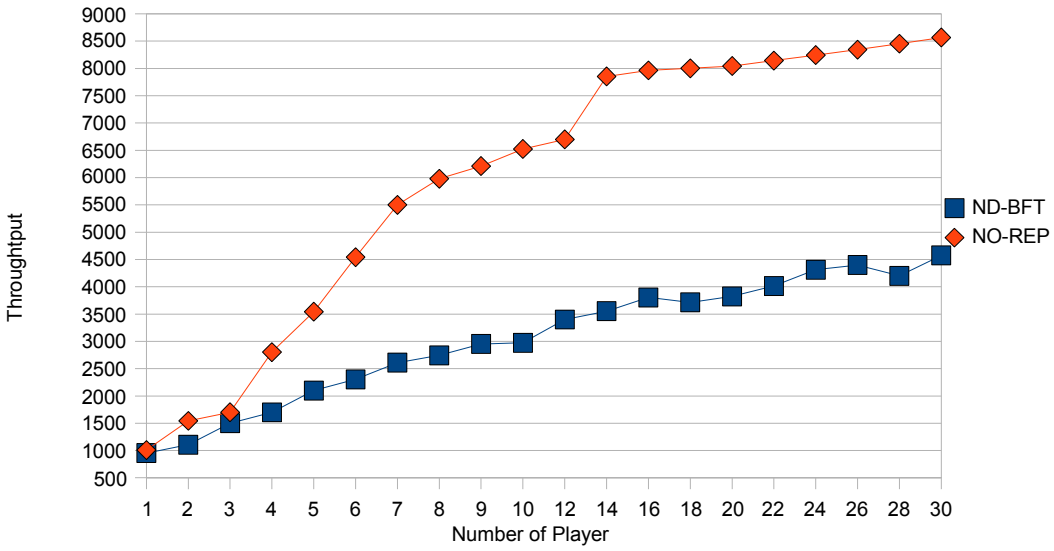

Figure 14: Throughput for online poker game (4 replicas)

Figure 15 presents the throughput measured with seven replicas. The average throughputs of both mechanisms are lower than the mechanism in previous experiment due to the number of replicas is increased. However, it might be surprising to find out that the throughput performance for seven replicas ND-BFT are very close to the four replicas ND-BFT. This could be helpful information for software designers because they can increase the security of their system without degrade the performance significantly. As the number of players increased, the throughput performance of NO-REP is increased by approximate 10% to 35% higher than ND-BFT.

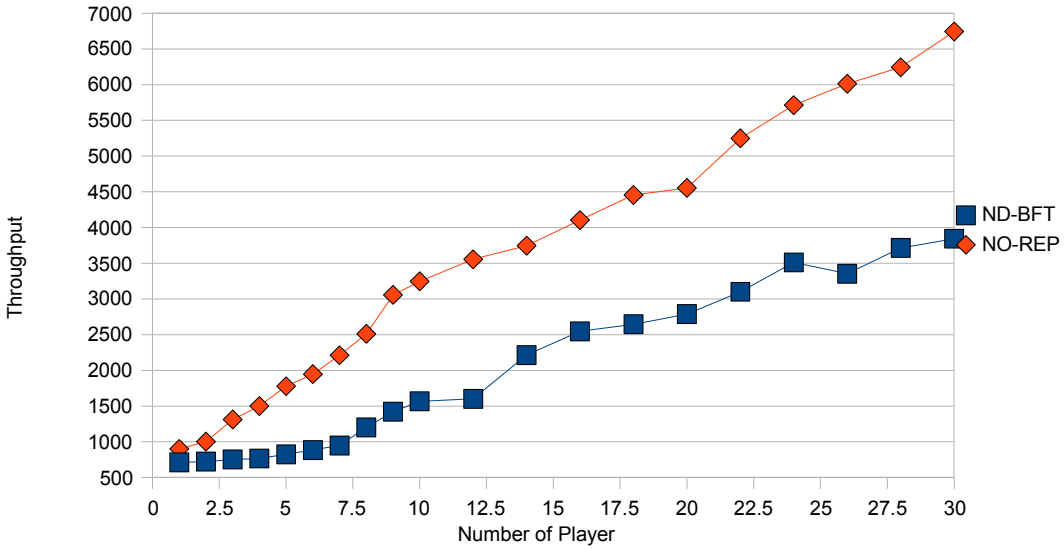

Figure 15: Throughput for online poker game (7 replicas)

There are two conclusions we gain from the experiment. First, our current mechanism would be more appropriate to be applied on the games which have small number of players. And there are more optimization works need to be done to improve the performance of the mechanism to be able to survive in large game which have considerable asynchronous network players. Secondly, the result shows that improving the resilience of the system by increasing the number of replica from four to seven does not degrade performance significantly.

CHAPTER V

RELATED WORKS

There is a vast body of research in the areas of fault tolerance and state machine replication. We present a brief overview of replication protocols to tolerate Byzantine fault for on practical istic applications. Our main focus, however, is on handling replica nondeterminism problem for Byzantine-fault-tolerant state machine replication protocols that provide support for general operations in an asynchronous environment.

Replica nondeterminism has been studied extensively under the benign fault model. However, there is no systematic classification of the common types of replica nondeterminism, therefore less attention has been payed on handling such non-determinism. [7] did provide a classification of some types of replica nondeterminism. However, they largely focused on the types of wrappable nondeterminism and

verifiable pre-determinable nondeterminism, except for nondeterminism caused by asynchronous interrupts, which we do not address in this work.

The replica nondeterminism caused by multithreading has been studied separately from other types of nondeterminism, again, under the benign fault mode only, in[5, 6, 11, 15]. However, these studies provided valuable insight on how to approach the problem of ensuring the consistency of replicated multithreaded applications. It is realized that what matters in achieving replica consistency is to control the ordering of different threads on access of shared data. The mechanisms to record and to replay such ordering have been developed. So do those for checkpointing and restoring the state of multi-threaded applications (for example,[21]). Even though the se mechanisms alone are not sufficient to achieve Byzantine fault tolerance for multithreaded applications, they can be adapted and used towards this goal. In this thesis, we have shown 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 under the Byzantine fault model.

Under the Byzantine fault model, the main effort on the subject of replica non-determinism control so far is to cope with wrappable and verifiable pre-determinable replica nondeterminism. In[17, 18], Castro and Liskov provided a brief guideline on how to deal with the type of nondeterminism that requires collective determination of nondeterministic values. The guide is very important and useful, as we have followed in this work. However, the guideline is applicable to only a subset of the problems we have addressed.

CHAPTER VI

CONCLUSION AND FUTURE WORKS

The growing reliance of our society on computer demands highly-available systems that provide correct service without interruptions. Byzantine faults such as software bugs, operator mistakes, and malicious attacks are the major cause of service interruptions. Byzantine fault tolerant algorithms have been invented to handle Byzantine faults by replicating servers and making them working in the same order. Replica nondeterminism, a problem that would disrupt the consistency of replica does not be addressed in Byzantine fault tolerant algorithms. Therefore there are no appropriate ways to handle such problem which is obtained by the majority of practical applications. This issue must be handled to ensure the total ordering of a Byzantine fault tolerant system.

This thesis presented a classification of common types of replica nondeterminism, and the mechanisms to handle them in the context of Byzantine fault tolerance. We also described how to integrate our mechanisms into a well-known BFT framework. Furthermore, we conducted extensive experiments to evaluate the performance of the BFT framework extended with our mechanisms and, for the first time, replicate a real online application, online poker game with our library-ND-BFT.

This chapter presents a summary of the main results in the thesis and direction for future works.

6.1 Summary

This thesis describes ND-BFT, a state-machine replication algorithm that based on Byzantine fault tolerant algorithm that handles replica nondeterminism problems occurred during the toleration of Byzantine faults.

BFT algorithms highly reply on replica consistency. BFT is the first state-machine replication algorithm that works correctly in asynchronous systems with Byzantine faults, in addition, it guarantees liveness provided message delays are bounded eventually, which require all replica execute the operation in the same order. It is a bad assumption that all replicas are deterministic, for instance, some services are data and time-last-modified which are set by reading the server's local clock, and if each server is in different location, the consistency of the whole system would diverge. Therefore, the mechanism to handle such behavior is necessary and needed.

It is also bad to assume that the replica nondeterminism can be treated in the same way. In the research of BFT, Castro and Liskvo simply treat the replica

nondeterministic problem by having the primary select the nondeterministic value independently or based on values provided by the replicas. The mechanism which categorized as wrappable nondeterminism and verifiable pre-determinable nondeterminism is indeed adequate for some services such as NFS. However, to provide our services in all practical application, a systematic categorization of all replica nondeterministic behavior is highly desired.

In this thesis, we categorized the replica nondeterministic into four types: VPRE, NPRE, VPOST and NPOST. ND-BFT, a generic program library with a simple interface, is based on BFT to provide a complete solution to each type of replica nondeterminism to the problem of building real services that tolerate Byzantine faults. For example, it includes efficient techniques to garbage collection information, to transfer state to bring replica up-to-date, to retransmit messages, and to handle services with different type of replica nondeterminism. The thesis presents a real service that was implemented using the ND-BFT library: the first Byzantine-fault-tolerant application that could handle complex replica nondeterministic problems.

The ND-BFT library and the corresponding ND-BFT application perform well. For example, ND-BFT performs only 13% lower throughput than BFT library and ND-BFT poker game performs 24% lower throughput than the nonreplicated poker game. Considering the ND-BFT could mask nondeterministic software errors, which seems to be the most persistent since they are the hardest to detect, the performance reduction is really acceptable. In fact, we always encountered such a

software bug while running our system. Our algorithm was able to continue running correctly in the presence of such kind of failure.

Additionally, the benefit of our algorithm can be increased by taking steps to increase diversity. One possibility is to have the diversity in the execution environment: the replicas can be administered by different people; they can be in different geographic locations; can they can have different configurations, for instance, run schedulers with different parameters or run different combination of services, but the ordering of the system is consistent. Thus the service provided by the system is reliable and totally ordered.

6.2 Future Work

We want to conduct deeper research that focusing on improve resilience to software bugs and online services, since the increasing popularity of those services would definitely bring the attention of hackers who wish to take the advantage by hacking or sabotaging those services. Not only online gaming application such as Blackjack and Texas Hold'em, but also several independent implementations available of operating systems and important services (e.g., file systems, databases, and web servers), replicas can run different operating system and different implementations of the code for these services. It is necessary to implement a small software layer for this to work. This could be simplified by the using existing protocols to access important services. There are also some research works on how to make this layer works more efficiently.

It is possible to improve security by combining our algorithm with other existing Byzantine fault tolerance algorithm. For instance, there are some interesting issues on using threshold signature techniques to replace BFT algorithm. The adapted BFT algorithm consists of three communication rounds (under normal operation) for Byzantine agreement and an additional round run at the beginning for key shares distribution. The Byzantine agreement algorithm works similar to the BFT algorithm except the third round, where each replica generates a partial signature (using its key share) to sign the client's message and piggyback the partial signature to the Commit message. Each replica combines the partial signatures into a threshold signature. The signature is then mapped into a number to seed the PRNG. Despite the elegance of the threshold signature, the algorithm, however, might not be practical in the Internet environment. First of all, it depends on a trusted dealer at the beginning to generate a key pair, divide the private key into several key shares and it must also be responsible for distributing the key shares to all replicas. If the dealer is compromised, the entire system can be easily penetrated by the adversary. Meanwhile, the threshold signature is computationally expensive, especially when generating the threshold signature (for a 1024-bit threshold signature it usually takes 73.9ms on IBM xSeries 330 1U rackmount PC with 1.0GHz Pentium III CPUs, 1.5 GB EEC PC133 SDRAM, and two 36 GB IBM UltraStar 36LZX hard drives)(Rhea *et al.*, 2003). Furthermore, the validity on the use of the threshold signature as the seed to the PRNG remains to be proved secure.

This thesis focused on the performance of the ND-BFT library in the normal case. It is important to perform an experimental evaluation of the reliability and

performance of our library with faults by using fault-injection techniques. The difficulty is that attacks are hard to model. Ultimately, we would like to make a real service on Internet and develop the modules and tools to record, verify and replay nondeterministic values to evaluate ability of our algorithm.

BIBLIOGRAPHY

[1] L. Lamport, R. Shostak, and M. Pease, "*The Byzantine general problem*," ACM Transactions on Programming Languages and Systems, Volume 4, No. 3, pp.382-401, July 1982.

[2] M. Castro, and B. Liskov, "*Practical Byzantine fault tolerance and proactive recovery*," ACM Transactions on Computer Systems, Volume 20, No. 4, pp.398-461, November 2002.

[3] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "*Separating agreement from execution for Byzantine fault tolerant Services*," Proceedings of the ACM Symposium on Operating System Principles, Bolton Landing, NY, pp. 253-267, October 2003.

[4] B. Arkin, F. Hill, S. Marks, M. Schmid, and T.j. Walls. "*H Additionally, the benefit of our algorithm can be increased by taking steps to increase diversity. One possibility is to have diversity in the execution environment: the replicas can be administered by different people; they can be in different geographic locations; can they can have different configurations, for instance, run schedulers with different parameters or run different combination of services, but the ordering of the system is consistent. Thus the service provided by the system is reliable and totally ordered.ow we learned to check at online poker: A study in software security.* Http://www.developer.com/java/other/article.php/10936_616221_1," September 1999.

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

[6] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. "*Loose synchronization of multithreaded replicas*," Proceedings of the Internation Symposium on Repliable Distributed systems. pp. 250-255. Suita, Japan, October 2002.

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

[8] M. Castro and B. Liskov. "*Practical Byzantine fault tolerance*," Proceeding of the Third Symposium on Operating Systems Design and Implementation," New Orleans, February 1999.

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

[10] L. Moser and M. Melliar-Smith. "*Consistent asynchronous checkpointing of multithreaded application programs based on semi-active or passive replication*," US Patent Application No.200500304014, 2005.

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

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

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

[14] J. Viega and G. McGraw. "*Building Secure Software*," Addison-Wesley, 2002

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

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

[17] M. Castro and B. Liskov. "*Proactive recovery in a Byzantine fault-tolerant system*." Proceeding of the Third Symposium on Operating Systems Design and Implementation, San Diego, October 2000.

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

[19] M. Castro and B. Liskov. "*BASE: Using abstraction to improve recovery*," ACM Transactions on Computer Systems, 21(3): 236-269, August 2003.

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

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

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

[23] S. Forrest. "*Building diverse computer systems*," Proceeding of the 6th workshop on Hot Topics in Operating Systems, May 1997.

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "I*mpossibility of distributed consensus with one faulty process*," Journal of the ACM, 32(2): 374-382, April 1985.

[25] Y. Amir, C. Damilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "*Steward: Scaling by Byzantine fault-tolerant systems to wide area networks*," Tech. Rep. CNDS-2005-3, Johns Hopkins University and CSD TR 05-029, Purdue University, http://www.dsn.jhu.edu, December 2005.

[26] L. Lamport. "*Paxos made simple*," SIGACTN: SIGACT News(ACM Special Interest Group on Automata and Computability Theory), vol. 32, 2001.

[27] Y. G. Desmedt and Y. Frankel. "*Threshold Cryptosystems*," in CRYPTO '89: Proceeding of Advances in Cryptology, pp. 307-315. Springer-Verlag Newyork, Inc., 1989.

[28] A. Shamir. "*How to share a secret*," Commun, ACM, vol. 22, no. 11, pp. 612-613, 1979

[29] V. Shoup. "*Practical threshold signatures*," Lecture Notes in Computer Science, vol.1807, pp. 207-223, 2000.

[30] R. L. Rivest, A. Shamir, and L. M. Adleman. "*A method for obtaining digital signatures and public key cryptosystems*," Communications of the ACM, vol. 21, pp. 120-126, February 1978.

[31] F. B. Schneider. "*Implementing fault-tolerant service using the state machine approache: A tutorial*," ACM Computing Surveys, vol. 22, no. 4, pp. 299-319, 1990.

[32] P. Feldman. "*A Practical Scheme for Non-Interactive Verifiable Secret Sharing*," Proceeding of the 28th Annual Symposium on Foundations of Computer

Science, (Los Angeles, CA, USA), pp. 427-437, IEEE Computer Society, IEEE, October 1987.

[33] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. "*The Secure Ring protocols for securing group communication*," Proceeding of the IEEE 31st Hawaii International Conference on System Sciences, vol. 3, pp. 317-326, January 1998.

[34] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. "*Robust threshold dss signatures*," Inf. Coput., vol. 164, no. 1, pp. 54-84, 2001.

[35] D. Malhi and M. Reiter. "*Byzantine quorum systems*," Journal of Distributed Computing, vol. 11, no. 4, pp. 203-213, 1998.

[36] D. Malhi and M. Reiter. "*An architecture for survivable coordination in large distributed systems*," IEEE Transactions on Knowledge and Data Engineering, vol. 12, no. 2, pp. 187-202, 2000.

[35] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. "*HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance*," Apprearing in the 7th USENIX Symposium on Operating System Design and Implementation(OSDI), November 2006.