Design and Implementation of a Byzantine Fault Tolerance Framework for Non-Deterministic Applications*

Honglei Zhang and Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University, 2121 Euclid Ave, Cleveland, OH 44115
{h.zhang105,w.zhao1}@csuohio.edu

L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
{moser,pmms}@ece.ucsb.edu

August 9, 2010

Abstract

State-machine-based replication is an effective way to increase the availability and dependability of mission-critical applications. However, all practical applications contain some degree of non-determinism. Consequently, ensuring strong replica consistency in the presence of application non-determinism has been one of the biggest challenges in building dependable distributed systems. In this article, we propose a classification of common types of application non-determinism with respect to the requirement of achieving Byzantine fault tolerance, and present the design and implementation of a Byzantine fault tolerance framework that controls these types of non-determinism in a systematic manner.

Keywords: Byzantine Fault Tolerance, Intrusion Tolerance, Replica Non-Determinism, Security, Fault Tolerance Middleware

*An earlier version of this paper was presented at the IEEE International Symposium on Dependable, Autonomic and Secure Computing, Columbia, MD, 2007 [25].

1 Introduction

State-machine-based Byzantine fault tolerance (BFT) [6, 8, 9, 10] is a promising approach to increasing the dependability of mission-critical applications in the presence of malicious attacks and other types of faults. The BFT algorithms employed in such an approach require the replicas to operate deterministically, *i.e.,* given the same request under the same state, all replicas of the server produce the same reply and transition to the same state. However, all practical applications contain some degree of non-determinism. When such applications are replicated to achieve fault and intrusion tolerance, their non-deterministic operations must be controlled to ensure replica consistency. Furthermore, unique to the Byzantine fault model, some types of non-determinism, if not properly handled, can be exploited to compromise the integrity of the services provided.

To the best of our knowledge, only the most simplistic types of non-determinism have been handled for the Byzantine fault model [6, 8, 9, 10], which we term *wrappable non-determinism* and *verifiable pre-determinable non-determinism*. The former means that the non-deterministic operation and its side-effects can be mapped into some pre-specified abstract operations and state, which are deterministic. The latter means that non-deterministic values can be determined prior to the execution of a request, and the values proposed by one replica can be verified by other replicas in a deterministic manner, and accepted if they are believed to be correct.

The mechanisms designed to handle these types of non-determinism [6, 8, 9, 10] either are not effective in guaranteeing replica consistency and/or are not effective in masking Byzantine faults, if the application to be replicated exhibits other types of non-deterministic behavior. For example, many online gaming applications contain non-deterministic values (*e.g.,* random numbers that determine the state of the applications) that are proposed by one replica but cannot be verified by another replica. It is dangerous to treat this type of non-determinism in the same way as verifiable pre-determinable non-determinism because a faulty replica could use a predictable algorithm to update its internal state and collude with its clients without being detected, which would defeat the purpose of applying existing BFT mechanisms. As another example, multi-

threaded applications exhibit non-deterministic values (*e.g.,* thread interleaving) that cannot be determined prior to the execution of a request (without losing concurrency), which cannot be handled by existing BFT mechanisms.

This paper makes several contributions:

- We introduce a classification of common types of replica non-determinism present in many applications. The classification is based on two criteria: whether or not the non-deterministic operations can be determined prior to the execution of a request, and whether or not the values associated with the non-deterministic operations sent by one replica can be verified by another replica. This classification leads to four types of non-determinism: verifiable pre-determinable non-determinism, non-verifiable pre-determinable non-determinism, verifiable post-determinable non-determinism, and non-verifiable post-determinable non-determinism.

- We describe a set of mechanisms that can be used to control these types of non-deterministic operations. To cope with non-verifiable pre-determinable non-determinism, we propose to rely on the collective input from a quorum of replicas, and a Byzantine agreement [15] step to ensure that all non-faulty replicas agree on the same set of values. To cope with non-verifiable post-determinable non-determinism, we propose to launch a monitoring process prior to the execution at a backup and to compare the generated reply message with the one supplied by the primary.

- We have implemented the mechanisms and integrated them into a well-known BFT framework [6, 8, 9, 10]. We present the results of our performance evaluation of the working prototype, which show that our mechanisms introduce very moderate runtime overhead.

2 Background

In this section, we present background information related to this research. First, we describe the fault models, then we cover several replication techniques commonly used to achieve fault tolerance under various

fault models, together with the concept of strong replica consistency. Next, we discuss how the fault tolerance mechanisms are typically positioned in practice. Finally, we focus on the concept of Byzantine fault tolerance and present a popular Byzantine fault tolerance algorithm.

2.1 Fault Models

Faults can be categorized as follows [23]:

- *Crash fault*: A crash fault occurs when a component of a system operates correctly up to some point in time, after which it produces no further results. For example, when the power is lost, a crash fault occurs for any process running on the host.
- *Omission fault*: An omission fault occurs when a component produces some results but not others. Message loss is an example of an omission fault.
- *Timing fault*: A timing fault occurs when a component produces results at the wrong time, either too early or too late.
- *Commission fault*: A commission fault occurs when a process generates incorrect results. A commission fault in which a process generates incorrect results that are intentionally designed to mislead the algorithms or other components of the system is an example of a Byzantine fault.

In general, the *crash fault model* refers to a model in which a process is subject to crash, omission, and timing faults, but not commission faults. The *benign fault model* refers to a model in which a process is subject to non-malicious commission (*i.e.,* non-Byzantine) faults, in addition to crash, omission and timing faults. The *Byzantine fault model* refers to a model in which a process is subject to arbitrary types of faults.

2.2 Replication Techniques and Strong Replica Consistency

The basic strategy to protect an application against faults is replication, so that if one replica becomes faulty, another replica is available to provide the service. However, with replication comes a challenge:

how to ensure that all non-faulty replicas have consistent state. Maintaining strong replica consistency is important because, otherwise, the integrity of the system might be compromised. An example is shown as part of the discussion on active replication below.

Common replication techniques, and mechanisms for ensuring strong replica consistency with respect to these replication techniques, are the following:

- *Active replication*: Active replication is also referred to as state-machine replication [19]. In active replication, all replicas receive the client's request, process it, generate and send back the reply to the client. To maintain strong replica consistency, all requests must be delivered in a total order to each replica. To understand why the total ordering of requests is necessary, consider the following on-line auction example. The server, which runs the on-line auction application, is replicated with two replicas R1 and R2. Two clients, C1 and C2, are trying to outbid each other for an item (assuming only one such item is available). At the closing moment of the auction, C1 and C2 each place a bid with the same price concurrently. If at R1, the bidding request m1 from C1 is ordered ahead of the bidding request m2 from C2, C1 would be declared the winner of the auction by R1. On the other hand, if at R2, m1 and m2 are ordered differently (*i.e.,* m2 is ahead of m1), C2 would be declared the winner by R2. If this happens, the state of the two replicas diverge and the integrity of the auction application is compromised, which might require lengthy manual resolution. If m1 and m2 had been totally ordered, then either C1 would have been declared the winner by both replicas (if m1 is ordered ahead of m2), or C2 would have been declared the winner by both replicas (if m2 is ordered ahead of m1).

  Active replication requires that each replica operates deterministically. In the presence of non-deterministic operations, inter-replica coordination is often needed to ensure that all replicas use exactly the same set of values for the non-deterministic operations, similar to the total ordering requirement for requests.

- *Passive replication*: In passive replication [17], one of the replicas is designated as the leader, referred to as the primary, and the remaining replicas are backups. Only the primary processes the client's request, and sends back the reply to the client. To ensure strong replica consistency in the presence

of replica non-determinism, the primary must send the update of its state to the backups prior to the sending of the reply, or it must send both the reply and the state update atomically to all replicas and the client [17].

- *Semi-active replication*: Semi-active replication is a variation of active replication [17]. One of the replicas is designated as the primary, and the remaining replicas are designated as backups. Even though all replicas process the client's request, only the primary sends back the reply to the client, and most importantly, the primary determines the total ordering of the request, records its decisions for all non-deterministic operations (if present), and multicasts the total order and decision data on the non-deterministic operations to the backups, which use that information to direct their own executions.

- *Semi-passive replication*: Semi-passive replication is a variation of passive replication [12] in which the primary communicates state updates to the backups for each operation. The backups update their states, but do not perform the operations and do not produce outgoing messages. Semi-passive replication aims to reduce the cost of recovery in the case of primary failure.

To provide Byzantine fault tolerance, we use active replication with an agreement algorithm based on a quorum and with a primary that receives messages from the clients and multicasts messages to the backups, which return the results to the clients.

2.3 Positioning of Fault Tolerance Mechanisms

Regardless of the replication technique used, it is common practice to implement the fault tolerance mechanisms in a middleware layer sandwiched between the application (on both the server and the client sides) and the operating system, as shown in Figure 1. The main advantage of this practice is that it cleanly separates the application logic and the fault tolerance logic. The application interacts with the fault tolerance mechanisms via a predefined set of application programming interfaces [6] or transparently through library inter-positioning [26].

All remote interactions between the client and the replicated server have to go through the middleware

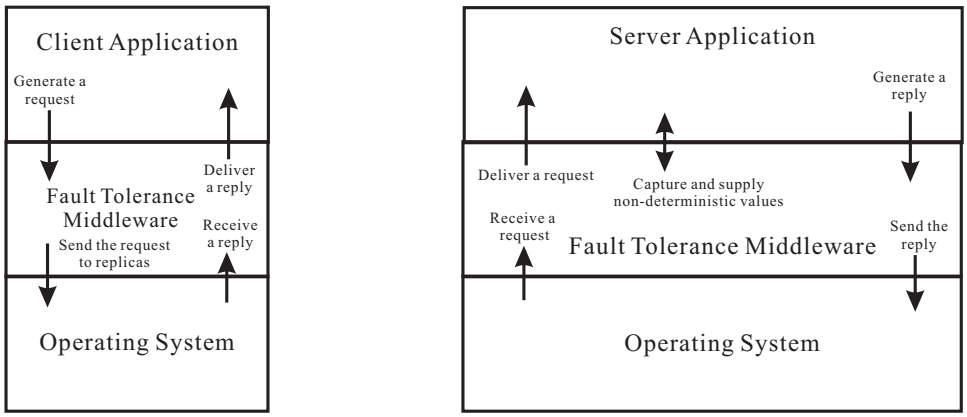

Figure 1: A common practice is to implement the fault tolerance mechanisms in a middleware layer sandwiched between the application and the operating system.

layer. To control replica non-determinism, all interactions between the application and the operating system are mediated by the middleware layer as well.

A request sent by a client to a server replica is first *received* at the middleware layer. Then a total order is imposed on the message by the fault tolerance mechanisms, after which the request is *delivered* to the application if all previous requests have been delivered. Similarly, any non-deterministic value is first obtained/decided by the fault tolerance mechanisms and subsequently supplied to the application at the appropriate time.

2.4 Byzantine Fault Tolerance

Byzantine fault tolerance (BFT) refers to the capability of a system to tolerate Byzantine faults. Replication techniques that rely on the input of a single replica, such as passive replication, semi-active replication, and semi-passive replication, are not appropriate for Byzantine fault tolerance, because that replica, if it is Byzantine faulty, might disseminate conflicting information to other replicas to cause replica inconsistency. Therefore, active replication must be used.

The most well-known BFT algorithm is due to Castro and Liskov [6]. The BFT algorithm is designed to support client-server applications running in an asynchronous distributed environment under the Byzantine

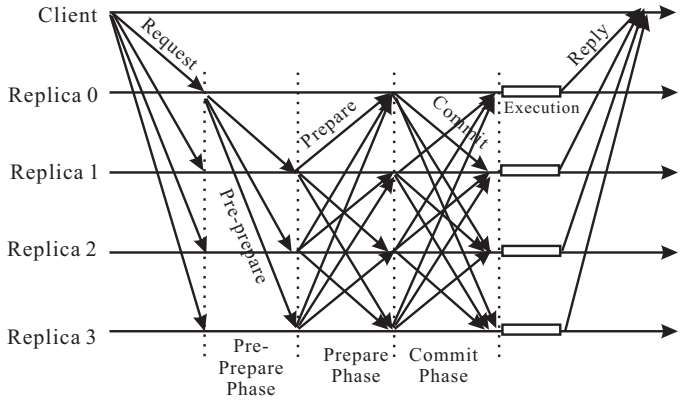

Figure 2: Normal operation of the BFT algorithm.

fault model. The BFT algorithm requires the availability of $3f + 1$ replicas to tolerate up to f Byzantine faulty replicas. The value of f that should be used depends on the risk the system faces and the desired degree of system reliability. It is conceivable that $f = 1$ would suffice for many applications. Among the $3f+1$ replicas, one of them is designated as the primary while the remaining replicas are backups. Note that the primary does not enjoy the total authority as it does in the replication techniques designed to cope with crash faults. The input from the primary is checked by other non-faulty replicas. If the primary is found to have disseminated wrong or conflicting messages (or no message at all for a sufficiently long duration), the primary is demoted to the role of a backup and another replica is selected to replace it using a round-robin scheme. This process is called a view change.

According to the BFT algorithm, the request issued by the client is captured by the fault tolerance mechanism in the middleware layer, and is sent to the primary, which is responsible to orchestrate the total ordering process. The mechanism at the client side collects the corresponding replies from the server replicas until it receives a consistent reply from $f + 1$ server replicas, at which time it delivers the reply to the client application. This mechanism ensures that the reply must have been generated by a non-faulty replica because there are at most f faulty replicas (according to the assumption). Because the primary could be faulty, the request is multicast to all replicas if a consistent reply cannot be obtained from $f + 1$ replicas within a reasonable time period.

As shown in Figure 2, the normal operation of the BFT algorithm involves three phases. During the first phase (called the pre-prepare phase), the primary multicasts a pre-prepare message to the backups containing the client's request, the current view number and a sequence number assigned to the request. A backup verifies the request message and the ordering information. If the backup accepts the message, it multicasts to all other replicas a prepare message containing the ordering information and the digest of the request being ordered. This starts the second phase, *i.e.,* the prepare phase. A replica waits until it has collected $2f$ prepare messages from different replicas (including the message it sent if it is a backup) that match the pre-prepare message before it multicasts a commit message to other replicas, which starts the third phase (*i.e.,* commit phase). The commit phase at a replica concludes when the replica has received $2f$ matching commit messages from other replicas. At this point, the request message has been totally ordered (*i.e.,* all non-faulty replicas have reached Byzantine agreement [15]) and the message can be delivered to the server application if all previous requests have already been delivered.

3 Related Work

Replica non-determinism for the crash fault model [2, 3, 4, 5, 14, 16, 17, 20, 24, 26] has been studied extensively. However, there is no systematic classification of common types of replica non-determinism, and even less so for the unified handling of such non-determinism. Classifications of some types of replica non-determinism are provided by [4, 5, 17]. However, those types of non-determinism largely fall within the types of wrappable non-determinism and verifiable pre-determinable non-determinism, with the exception of the non-determinism caused by asynchronous interrupts, which we do not address in this work.

The replica non-determinism caused by multi-threading has been studied separately from other types of non-determinism, again, under the crash fault model in [2, 3, 14, 16, 18, 26]. These studies provide valuable insight on how to approach the problem of ensuring consistent replication of multi-threaded applications. What matters in achieving replica consistency is to control the ordering of different threads in their access to shared data. The mechanisms to record and replay such ordering have been developed [26], as have those

for checkpointing and restoring the state of multi-threaded applications (*e.g.*, [13]).

As discussed in Section 2, a number of replication techniques, including passive replication, semi-active replication, and semi-passive replication [17, 12, 26], have been developed to cope with replica non-determinism under the crash fault model. In those replication techniques, the primary decides on the total ordering of messages and non-deterministic values, and such decisions are not verified by the backups. These techniques are not applicable in the presence of Byzantine faults because a faulty primary could send conflicting or wrong decisions to the backups, which would lead to the divergence of replica state. Nevertheless, the previous research provides great insight on what operations can lead to replica non-determinism, and how to record and replay non-deterministic operations.

For the Byzantine fault model, the main effort of controlling replica non-determinism control thus far is to cope with wrappable and verifiable pre-determinable replica non-determinism [6, 8, 9, 10]. In [6], Castro and Liskov provide brief but important and useful guidelines on how to deal with the type of non-determinism that requires collective determination of the non-deterministic values. Those guidelines, however, are applicable to only a subset of the problems that we address.

4 Classification of Replica Non-Determinism

We distinguish replica non-determinism into the following three major categories:

- *Wrappable non-determinism*: This type of replica non-determinism is easily controlled by using an infrastructure-provided or application-provided wrapper function, *without explicit inter-replica coordination.* For example, 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 [10].

- *Pre-determinable non-determinism*: In this type of replica non-determinism, the values are known *prior to* the execution of a request. This type of replica non-determinism requires inter-replica coordination

to ensure replica consistency. For example, it is possible to know that a random number will be needed during the execution of a request (*e.g.,* from the specification of the remote method) and the BFT mechanisms can decide which random number to use prior to the execution.

- *Post-determinable non-determinism*: In this type of replica non-determinism, the values are recorded only *after* the request is submitted for execution and the non-deterministic values won't be known until the end of the execution. This type of replica non-determinism also requires inter-replica coordination to ensure replica consistency. For example, it is virtually impossible to predefine the thread interleaving for a multi-threaded application prior to execution. The only practical way is to record such interleaving at the primary and enforce the same interleaving at the backups.

In this paper, we will not discuss wrappable non-determinism further, because it can be dealt with using a deterministic wrapper function without inter-replica coordination, and also because it has been thoroughly studied [10]. Instead, we focus on the remaining two types of replica non-determinism.

Based on whether or not a replica can verify the non-deterministic values proposed (or recorded) by another replica, replica non-determinism can be further classified into the following types:

- *Verifiable non-determinism*: In this type of replica non-determinism, the values can be verified by other replicas. The verification is done by comparing each value associated with the non-deterministic operation of one replica with that of another replica. Obviously, for a non-deterministic operation, it is impossible to expect that the two values are identical. For the purpose of verification, a heuristic bound on the differences in the values must be predetermined or dynamically adjusted. If the bound is estimated incorrectly, a backup might mistakenly suspect the primary due to the out-of-bound value proposed by the primary, which might lead to an unnecessary view change. However, the safety property of the system will not be violated because of the mistake.

- *Non-verifiable non-determinism*: In this type of replica non-determinism, the values cannot be completely verified by other replicas. Online gaming applications, such as Blackjack [1] and Texas Hold'em [21], exhibit this type of non-determinism. The integrity of services provided by such applications depends

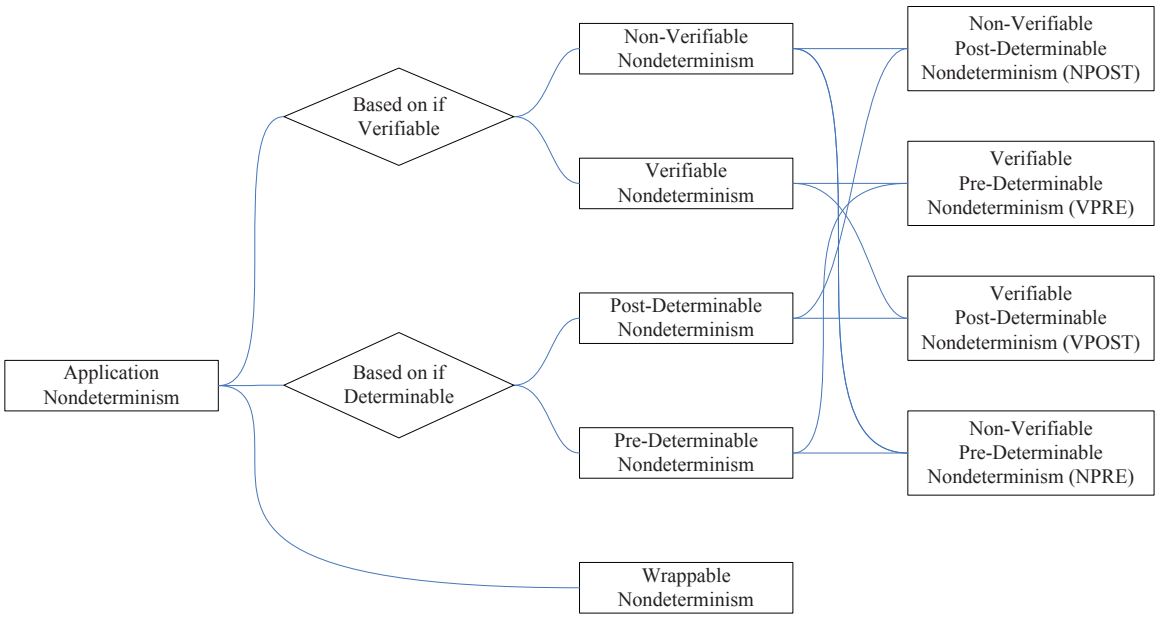

Figure 3: A classification of common types of application non-determinism.

on the use of secure random number generators. For the best security, it is essential to make the choice of a random number unpredictable, which prevents another replica from verifying it.

Overall, as summarized in Figure 3, our classification yields four types of replica non-determinism: verifiable pre-determinable non-determinism (VPRE), non-verifiable pre-determinable non-determinism (NPRE), verifiable post-determinable non-determinism (VPOST), and non-verifiable post-determinable non-determinism (NPOST).

In Figure 4, we provide an example of each type of replica non-determinism except VPOST, because we have yet to identify a commonly used application that exhibits this type of non-determinism. Figure 4 also includes an analysis of the risk of not controlling the non-determinism and a synopsis of the solution for each type.

In practical applications, the execution of a request often involves more than one type of non-determinism, for example, both time-related non-determinism (which is of the verifiable pre-determinable type) and multi-threading-related non-determinism (which is of the non-verifiable post-determinable type). Thus, considering the possibility of composite types of non-determinism, there can be 12 different combinations of non-

determinism types. Because we have yet to identify the VPOST non-determinism in practical applications, we only consider the following seven types of non-determinism:

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

5 Controlling Replica Non-Determinism for BFT

In this section, we present the system model, application program interfaces (APIs), mechanisms for controlling different types of replica non-determinism for Byzantine fault tolerance, and informal proofs of correctness for our mechanisms.

5.1 System Model

We consider a client-server application operating in an asynchronous distributed environment. To achieve liveness, it is necessary to assume a certain degree of synchrony. Like [9], we assume that the message transmission time and the processing time have an asymptotic upper bound.

Example Pseudo-Code	Non-Deterministic Operations and Risk Analyses	Synopsis of Solutions

VPRE Example
`// in a network file system`
`void updateFile`
`(FileHandle fh, Content c)`
`{`
`retrieve file fh;`
`update file with content c;`
`time_t t = gettimeofday();`
`fn.last_modified_time = t;`
`}`
	`gettimeofday()` in this example is a verifiable pre-determinable non-deterministic operation.
	
	If not controlled, this operation can cause the same file containing different metadata (for the last modified timestamp) at different replicas, which will lead to divergent states at the replicas.
		The primary proposes a timestamp to be used for the `last_modified_time`, and piggybacks it onto the request message to be ordered. A backup verifies the proposed timestamp using a heuristic bound, and suspects the primary, if the timestamp is out of bounds. A mistake in suspecting the primary will cause a view change, but will not put the safety of the system at risk.

NPRE Example
`// in an online game`
`void shuffleCards()`
`{`
`unsigned int seed =`
`generateSeed(uuid());`
`// seed random number generator`
`srand(seed);`
 	
`for (int i=0; i < 52)`
`{// assign a random rank`
`cards[i].rank = rand()%52;`
`}`
`// sort cards based on rank`
`cards = sortCards(cards);`
`}`
	`generateSeed()` in this example is a non-verifiable pre-determinable non-deterministic operation.
	
	If not controlled, this operation can lead to different shuffled cards by different replicas and, hence, divergent states at the replicas.
	
	One should never attempt to replace the random number by a number generated deterministically or from a low entropy source, because doing so can lead to an easy-to-predict hand.
		Each replica proposes its own value (i.e., the random number) and the primary selects the proposed values from 2f+1 replicas.
		
		The set of 2f+1 values is piggbacked onto the request message to be ordered, so that all non-faulty replicas agree on the same set.
		
		The final value is computed based on the set of values according to a deterministic algorithm.

NPOST Example
`// in a multi-threaded server`
`// code executed by thread T1`
`{`
`...`
`acquireLockA(); self.a = newA;`
`acquireLockB(); self.b = newB;`
`releaseLockA(); releaseLockB();`
`...`
`}`
`// code executed by thread T2`
`{`
`...`
`acquireLockB(); self.b = newB;`
`acquireLockA(); self.a = newA;`
`releaseLockB(); releaseLockA();`
`...`
`}`
	This example involves two thread T1 and T2, and two shared variables a and b, each protected by a lock. Threads T1 and T2 concurrently execute a piece of code in which T1 calls `acquireLockA()` and then `acquireLockB()` and T2 calls `acquireLockB()` and then `acquireLockA()`. The multi-threading operations are non-verifiable post-deterministic non-deterministic operations.
	
	If not controlled, the non-deterministic thread interleavings might lead to the following problem. One replica successfully executes all the code (e.g., T1 has called `acquireLockA()` and `acquireLockB()` before T2 calls `acquireLockB()`), whereas another replica runs into a deadlock (e.g., T1 calls `acquireLockA()` when T2 calls `acquireLockB()` and both are blocked at the next lock acquisition operation).
		The primary records the order in which a thread is granted a lock and disseminates the ordering information to the backups.
		
		To prevent a faulty replica from sending conflicting information to different replicas, a Byzantine agreement step is needed to ensure that all non-faulty replicas receive the same information.
		
		Before executing according to the recorded threading information, a backup first launches a monitoring process as a precautionary measure to deal with possible crash or deadlock situations.

Figure 4: Examples of common types of replica non-determinism with pseudo-code, explanations, risk analyses, and synopses of the solutions.

Both the client and the server can be Byzantine faulty, *i.e.,* they can exhibit arbitrary faults. To achieve Byzantine fault tolerance, the server is replicated with $3f + 1$ replicas to tolerate up to f faulty nodes. We assume that messages are protected by a digital signature or an authenticator [7] to ensure their integrity. We assume that the adversaries have limited computing power so that they cannot break the digital signatures or authenticators of non-faulty replicas.

Each replica is modeled as a state machine. The replica is required to run, or rendered to run deterministically. The state change is triggered by remote invocations of the methods offered by the replica. In general, the client first sends its request to the primary replica. The primary replica then broadcasts the request message to the backup replicas and also determines the execution order of the message. All non-faulty replicas must agree on the same set of request messages with the same execution order. In other words, the request messages must be delivered to the server application at all non-faulty replicas reliably in the same total order.

To achieve the total ordering of messages, we use the BFT algorithm [9]. In later sections, we describe how to integrate our mechanisms for controlling replica non-determinism into the BFT algorithm, so that non-faulty replicas agree on both the message ordering and the non-deterministic values.

5.2 Application Programming Interface

The BFT framework is implemented as a library to be linked into the application code (on both server and client sides). As shown in Figure 5, the client-server application and the BFT mechanisms (within the BFT library) interact via a set of Application Programming Interfaces (APIs). The APIs contain downcalls to be invoked by the application for a number of purposes, for example, to initialize the BFT library with appropriate parameters and callback functions, to start the event loop managed by the BFT library, and to send requests to the server replicas. The APIs also contain upcalls to be implemented by the application, so that the BFT mechanisms can deliver requests to the server application, retrieve and verify non-deterministic values (if applicable), and retrieve and restore application state.

The core upcall APIs used to control replica non-determinism are described below:

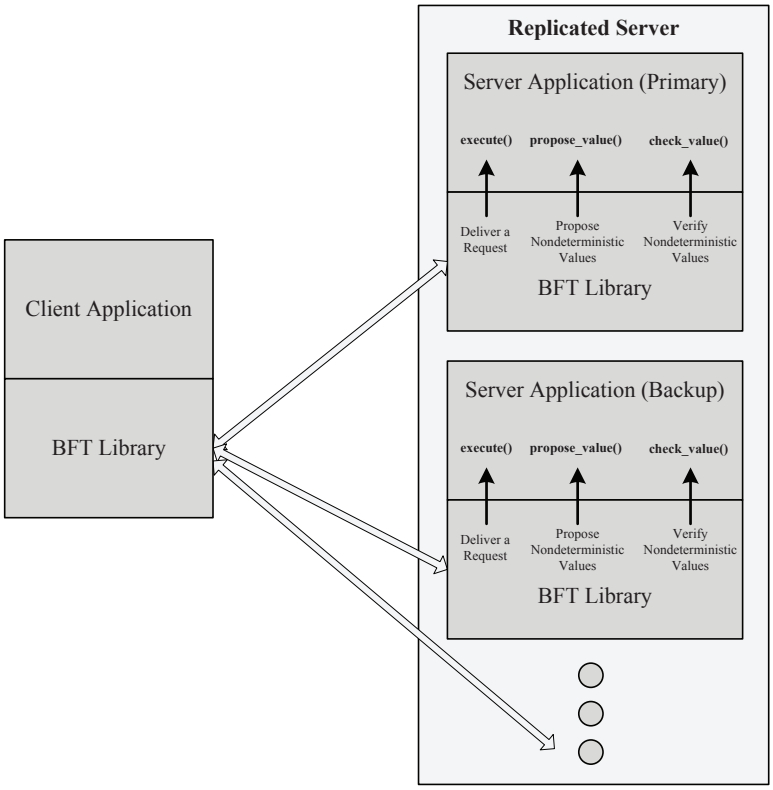

Figure 5: The positioning of the application and the BFT library, and the core interfaces between the two components.

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

This function is called when a replica wishes to find out the type of non-determinism involved in the request and to obtain its share of non-deterministic values (if applicable). In this API, `seqno` is the sequence number assigned to the client's request, `req` is a pointer to the request message, `ndet_type` is a pointer to the type of non-determinism the replica might exhibit when executing the request, and `ndet` is a pointer to the buffer that stores the non-deterministic values. This function returns appropriate values to indicate if the call is successful. Both `ndet_type` and `ndet` are out parameters, which means the application is expected to set their values.

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

This function is invoked when a backup replica wants to verify the type of non-determinism and the non-

deterministic values received from the primary replica (if applicable). The parameters are the same as those for the `propose_value()` function. The only difference is that `ndet_type` and `ndet` are now used as in parameters, which means that the information is passed to the application. The verification result is returned to the caller in the return value.

```
int execute(Byz_req *req, Byz_rep *rep, Byz_buffer *ndet, int cid, bool ro);
```

This function is called to deliver a request message to the application, together with the non-deterministic values. For operations with post-determinable non-determinism, this function is called by the primary to retrieve the recorded non-deterministic values. In this API, `req` is a pointer to the request message, `rep` is a pointer to the reply message to be generated by the replica, and `ndet` is a pointer to the non-deterministic values. The `ndet` parameter is an in-out parameter. Depending on the type of replica non-determinism, it might be an in parameter, which means that it points to the buffer that stores the non-deterministic values to be used by each replica, or an out parameter when a replica has post-determinable non-determinism and the function is invoked at the primary replica.

5.3 BFT Mechanisms

The Byzantine fault tolerance mechanisms work as follows. When the primary receives a client's request, if it is ready to order the message, it invokes the `propose_value()` callback function registered by the application. The application supplies the type of non-determinism involved in the execution of the request and, if applicable, the non-deterministic values.

The original BFT algorithm is extended with two communication phases, namely, the pre-prepare-update phase and the post-commit phase. In each phase, a new control message referred to by the phase name is introduced. The `pre-prepare-update` message is used in the additional phase for the replicas to reach Byzantine agreement on the collection of non-deterministic values contributed by different replicas when non-verifiable pre-determinable non-determinism is present. The `post-commit` message is used in the additional phase for the replicas to reach Byzantine agreement on the non-deterministic values recorded by the primary

replica after it has executed a request message (hence, the name post-commit when post-determinable non-determinism is present.)

In the following subsubsections, we provide detailed descriptions of the mechanisms for controlling each single type of non-determinism. The handling of composite types is straightforward. Using the same example given in Section 4, the time-related non-deterministic values can be determined during the pre-prepare-update phase, and the multi-threading-related non-determinism can be resolved in the post-commit phase. Note that to cope with composite types of replica non-determinism, the data structure used to store the non-deterministic values does not need to be made more sophisticated because it is the application's responsibility to generate and interpret the non-deterministic values, and for the same reason, there is no need to change the APIs.

5.3.1 Controlling VPRE Non-Determinism

If the non-determinism for the operation at the primary is of the type `VPRE`, the application provides the non-deterministic values in the `ndet` parameter. The obtained information is included in the `pre-prepare` message, and the message is multicast to the backup replicas.

On receiving the `pre-prepare` message, a backup replica invokes the `check_value()` callback function. The replica passes the information received regarding the non-determinism type and data values to the application, so that the application can verify that (1) the type of non-determinism for the client's request is consistent with what is reported by the primary, and (2) the non-deterministic values proposed by the primary is consistent with its own values. If either check is false, the `check_value()` call returns an error code, the backup replica then suspects the primary. Otherwise, the backup replica accepts the client's request and the ordering information specified by the primary, logs the `pre-prepare` message and multicasts a `prepare` message to all other replicas. From now on, the algorithm works the same as the original BFT algorithm, with the exception that the `prepare` and `commit` messages also carry the digest of the non-deterministic values. The normal operation of the modified BFT algorithm is illustrated in Figure 6.

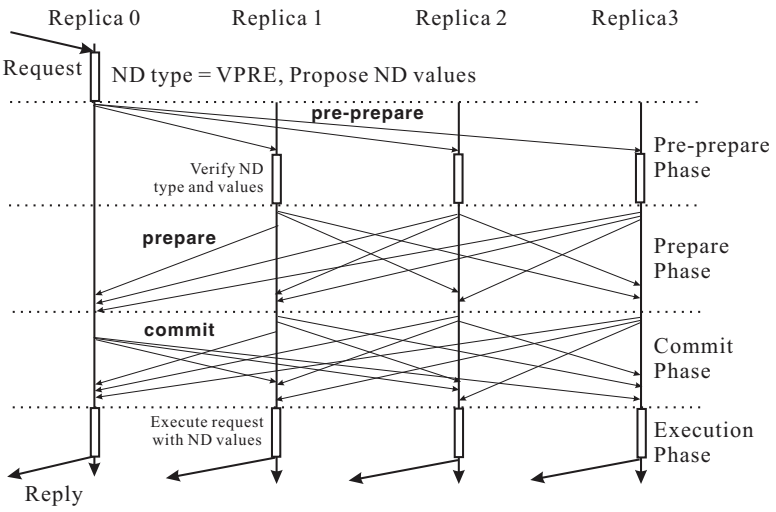

Figure 6: Normal operation of the modified BFT algorithm in handling verifiable pre-determinable non-determinism.

5.3.2 NPRE Non-Determinism

If the non-determinism for the operation at the primary is of the type `NPRE`, the application at the primary proposes its non-deterministic values. The type of non-determinism and the non-deterministic values are included in the `pre-prepare` message, and the message is multicast to all backup replicas.

On receiving the `pre-prepare` message, a backup replica invokes the `check_value()` callback function to verify the non-determinism type supplied by the primary replica (after it has verified the client's request and the ordering information). If the verification is successful, the backup replica invokes the `propose_value()` function to obtain its own non-deterministic values. It then builds a `pre-prepare-update` message including its own non-deterministic values, and sends the message to the primary.

When the primary receives 2f `pre-prepare-update` messages from different backup replicas (for the same client request), it builds a `pre-prepare-update` message, including the 2f+1 sets of non-deterministic values, each protected by the proposer's authenticator. The `pre-prepare-update` message itself is further protected by the primary's authenticator. The primary then multicasts the message to all backup replicas. From now on, the BFT algorithm operates according to the original algorithm, except that the `prepare` and

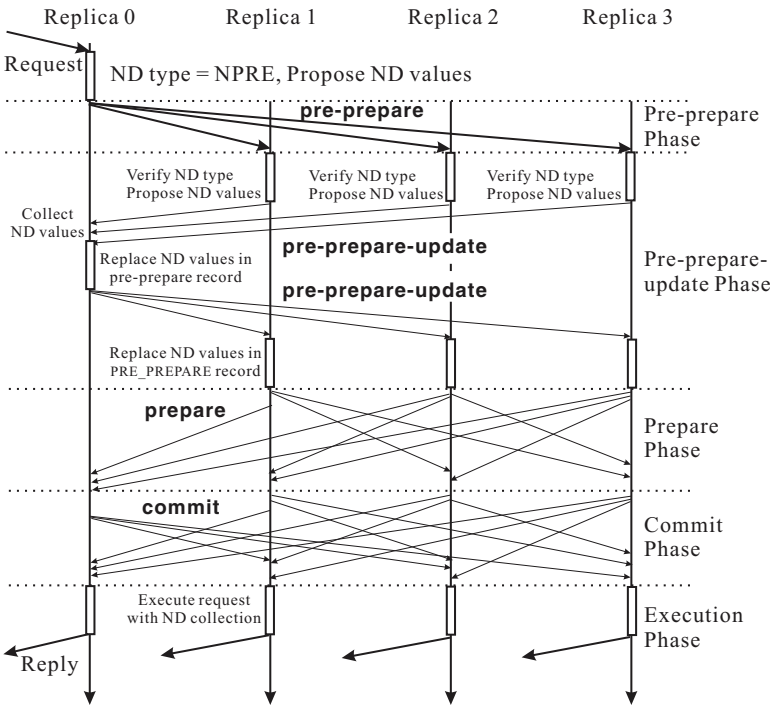

Figure 7: Normal operation of the modified BFT algorithm in handling non-verifiable pre-determinable non-determinism.

`commit` messages also carry the digest of the non-deterministic values, and the 2f+1 sets of non-deterministic values are delivered to the application as part of the `execute()` upcall. The normal operation of the modified BFT algorithm for this type of non-determinism is illustrated in Figure 7.

5.3.3 VPOST Non-Determinism

The normal operation of the modified BFT algorithm in handling this type of non-determinism is shown in Figure 8. The primary includes the non-determinism type (*i.e.,* `VPOST`) in the `pre-prepare` message without any non-deterministic values and multicasts the message to the backup replicas.

On receiving the `pre-prepare` message, a backup replica performs the `check_value()` upcall if it has verified the client's request and the ordering information. If the backup replica confirms the type of non-determinism, it proceeds to the commit phase as usual. Otherwise, the backup replica suspects the primary.

When the primary is ready to deliver the request message, it proceeds to perform the `execute()` upcall

and expects to receive both the reply message and the recorded non-deterministic values. Once the upcall returns, the primary stores the retrieved post-determined non-deterministic values, together with the digest of the reply, into a `postnd` log (to be sent to the backup replicas), and sends the reply message to the client. The digest of the reply is included in the `postnd` log, so that a backup replica can verify that the primary has actually used the non-deterministic values to generate the reply.

A post-commit phase is needed for the primary to disseminate the data in the `postnd` log to backup replicas and for all non-faulty replicas to ensure that they have received the same set of values for the corresponding request. Unlike the pre-prepare-update phase for controlling the `NPRE`non-determinism type, the post-commit phase involves all of the steps needed for non-faulty replicas to reach an agreement on the non-deterministic values, which requires three rounds of message exchange similar to those used to determine the ordering of the requests under normal operation. For the `NPRE`non-determinism type, the prepare and commit phases needed for the non-faulty replicas to reach Byzantine agreement on the non-deterministic values are integrated with those for the corresponding request message. We cannot do so for post-determinable non-determinism types because the ordering for the corresponding request has already been decided.

A backup replica does not deliver a request message until Byzantine agreement has been reached on the non-deterministic values for the request. If Byzantine agreement cannot be reached, or the verification of the non-deterministic values fails, a backup replica suspects the primary. Furthermore, when the backup replica produces a reply for the request, the digest of the reply is compared with that supplied by the primary. If the two do not match, the backup replica suspects the primary. Regardless of the comparison result, the backup replica sends the reply message to the client. It is safe to do so because the result is valid if all non-faulty backup replicas produce the same reply using the same set of non-deterministic values (even if they differ from the set actually used by the primary, which implies that the primary is lying and will be suspected).

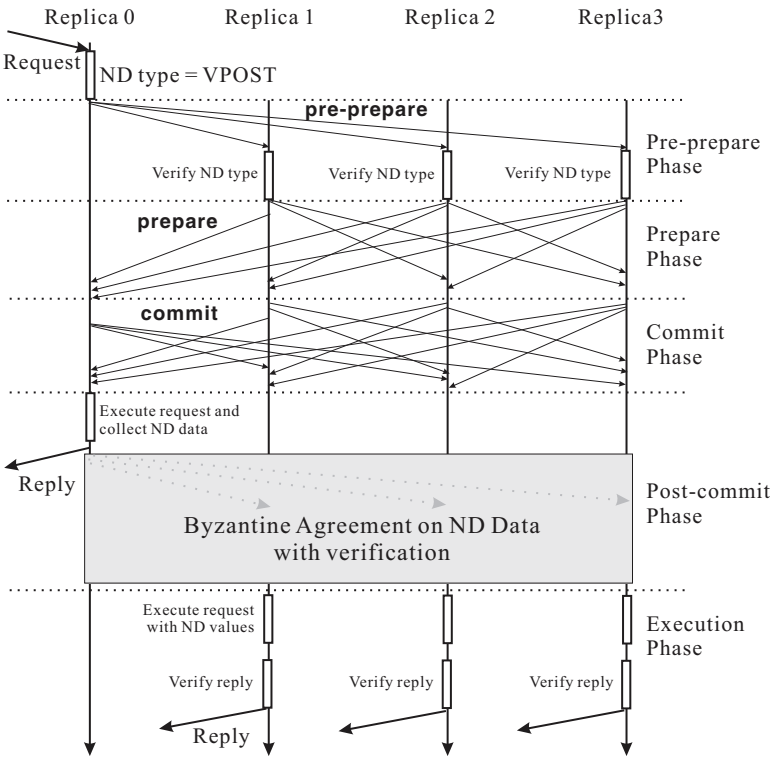

Figure 8: Normal operations of the modified BFT algorithm in handling verifiable post-determinable non-determinism.

5.3.4 NPOST Non-Determinism

The handling of non-verifiable post-determinable non-determinism involves the same steps as those described in the previous subsubsection until a backup replica is ready to deliver the request with the post-determined non-deterministic values, as shown in Figure 9.

The concern here is that a faulty primary could disseminate a set of incorrect non-deterministic values hoping either to confuse the backup replicas, or to block them from providing useful services to their clients. For example, if the non-deterministic values contain thread ordering information, a faulty primary can arrange the ordering in such a way that it leads to the crash of a backup replica (*e.g.,* if the attacker knows the existence of a software bug that leads to a segmentation fault), or it might cause a deadlock at a backup replica (the replica might perform deadlock analysis before it follows the primary's ordering to prevent this

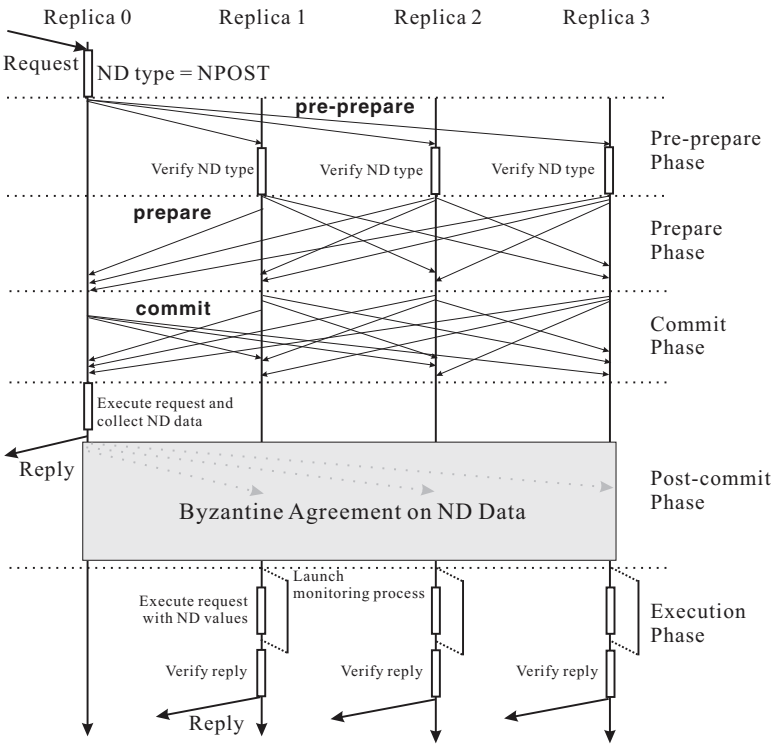

Figure 9: Normal operation of the modified BFT algorithm in handling non-verifiable post-determinable non-determinism.

from happening).

Because, in general, a backup replica cannot completely verify the correctness of the non-deterministic values until it executes the request, it is important for the replica to launch a separate monitoring process prior to invoking the `execute()` call. If the replica encounters a deadlock or a crash fault, the monitoring process can restart the replica and suspect the primary.

If it can successfully complete the `execute()` upcall, the backup replica performs the same reply verification procedure as that described in the previous subsubsection, and sends the reply to the client.

5.4 View Change

A faulty primary might prevent a non-faulty backup replica from reaching Byzantine agreement on the ordering of the application message and/or the associated non-deterministic values, in which case, a view

change becomes necessary. Moreover, it might take several view changes for a replica to reach Byzantine agreement, and different replicas might reach Byzantine agreement in different views. It is important to ensure that adequate information is propagated from one view to another so that if two replicas reach Byzantine agreement in different views, they agree on the same total ordering and the same non-deterministic values.

The view change mechanism for message ordering involves two control messages, *i.e.,* `view-change` and `new-view` messages. A non-faulty replica suspects the primary and initializes the view change if (1) its view change timer expires, or (2) it cannot verify the non-deterministic values (for verifiable non-determinism), or it generates a different reply (for non-verifiable non-determinism).

The view change message has the form $<\text{VIEW-CHANGE}, v+1, n, C, P, Q, PP, PQ, i>_{\sigma_i}$, where n is the sequence number of the latest stable checkpoint s known to replica i, C is the proof of s, i is the identifier of the sender, and P, Q, PP and PQ are defined as follows:

- P is the set of records for the requests that have been prepared at sender i in previous views.
- Q is the set of records for the requests that have been pre-prepared at sender i in previous views.
- PP is the set of records for the requests that have been post-prepared (on non-deterministic values) at sender i in previous views.
- PQ is the set of records for the requests that have reached the post-pre-prepared state at sender i in previous views.

The sets P and Q are required for requests. The sets PP and PQ are used to reach agreement on non-deterministic values for requests with post-determinable non-determinism. The sets P, Q, PP, and PQ have the same form $<d_m, d_{nd}, n, v, t>$, where d_m is the digest of the request, d_{nd} is the digest of the non-deterministic value, n is the sequence number, v is the view number, and t is the type of non-determinism.

A replica updates these sets of information immediately before sending the `view-change` message by using the records in its log. On sending the `view-change` message, the replica removes all `prepare`, `pre-prepare`,

`commit`, `post-pre-prepare`, `post-prepare`, `post-commit` messages from the log, because those messages are no longer useful.

When a replica receives a `view-change` message, it accepts the message, provided that all of the information in P, Q, PP and PQ is for view v or an earlier view. The replica sends a `view-change-ack` message to the primary in view $v+1$ if it accepts the `view-change` message. The `view-change-ack` message has the form <VIEW-CHANGE-ACK, $v+1$, i, j, d>σ_i where i is the sender id, j is the identifier of the sender of the `view-change` message being acknowledged, and d is the digest of the `view-change` message.

The primary in view $v+1$ collects each `view-change` message and the $2f$ corresponding `view-change-ack` messages, and stores them as an entry of an internal log S. Each entry of S is for a different replica. When the new primary receives $2f+1$ valid `view-change` messages, it constructs a `new-view` message using the information in S and multicasts the message to other replicas. The `new-view` message has the form <NEW-VIEW, $v+1$, V, X>σ_i, where V contains the proof of the view change and X contains the checkpoint that determines the starting state of the new view, and a set of requests, for each sequence number between h and $h+L$, with the associated non-deterministic values to reach agreement across views. (Here, h is the log's low watermark and L is the size of the log.) The checkpoint and the set of requests are updated each time the new information is added to S.

The primary in view $v+1$ chooses the checkpoint first from the information in S with the highest sequence number greater than h in the log of which at least $f+1$ non-faulty replicas from the set are known to be correct. Then, the primary chooses a request together with the associated non-deterministic values, if necessary, for each sequence number within the range h to $h+L$ to pre-prepare in the new view $v+1$. For pre-determinable types of non-determinism, the same request determination procedure is used as that of [9]. For post-determinable types of non-determinism, after the request is determined, a similar procedure is used to determine which non-deterministic values should be adopted. If no non-deterministic values can be adopted, a NULL value is included in the `new-view` message and the new primary will propose new values when the request is re-executed.

The primary in view $v+1$ updates its own state to reflect the contents of the message after sending

the `new-view` message. It fetches the state from other replicas if any requests, non-deterministic values, or checkpoints are missing. Then, it logs all requests as pre-prepared in $v + 1$.

When a backup in view $v + 1$ receives the `new-view` message, it verifies the message by checking the new view certificate V in the `new-view` message with the `view-change` messages it has collected. If a backup did not receive a `view-change` message from some replica included in V, it asks the primary to send it a proof of correctness that contains the original `view-change` message and $2f$ acknowledgments. The backup subsequently verifies the information by repeating the same procedure as that used by the primary to construct the `new-view` message. If the verification fails, the replica moves to another view immediately. Otherwise, it resumes normal operation for each request.

5.5 Proof of Correctness

We now provide an informal proof of correctness for our mechanisms. We argue here only for the correctness of the safety property of our mechanisms. For the liveness property, the correctness proof for the original BFT algorithm can be directly applied to our mechanisms.

Theorem 1. If a non-faulty replica delivers a request m with a sequence number n and a set of non-deterministic values nd in view v, then no other non-faulty replica delivers m with a different sequence number or a different set of non-deterministic values, and each non-faulty replica uses, or records (in the case of the primary), the same set of non-deterministic values for request m.

Proof: First, we prove that, if two non-faulty replicas deliver m in the same view v, then they also deliver the same set of non-deterministic values nd with m in the same total order.

For the `VPRE` type, the non-deterministic values are proposed by the primary and the agreement on the values is carried out with the request message itself. If non-faulty replicas agree on the ordering of the request message, they agree on the non-deterministic values as well. For the `NPRE` type, the non-deterministic values are collectively determined in the pre-prepare-update phase, and the consensus on the values is achieved by the three-phase Byzantine agreement algorithm. Again, if some non-faulty replicas commit the request m, they also agree on the associated non-deterministic values. For the `VPRE` and `NPRE` types, when the request

m is delivered at a non-faulty replica, the non-deterministic values that have been agreed upon are also delivered and used for execution.

For the `VPOST` and `NPOST` types, the agreement on the non-deterministic values among non-faulty replicas is guaranteed by the three-phase Byzantine agreement algorithm executed during the post-commit phase. When the request m is delivered at a non-faulty backup replica, the non-deterministic values associated with m are also delivered. The primary, if it is not faulty, must have recorded the non-deterministic values during its execution of m, and have disseminated the values to the backups during the post-commit phase. Therefore, the same set of non-deterministic values are used for execution at the primary (if it is not faulty) and the non-faulty backup replicas.

Next, we prove that the same statement is true if two non-faulty replicas deliver m in different views. Without loss of generality, we assume replica R_i delivers m in view v, and replica R_j delivers m in view w, where $w > v$.

Because R_i delivered m in view v, it must have committed m with a sequence number n. If the replica non-determinism associated with m is of the type `VPRE` or `NPRE` , then R_i must have also committed the set nd of non-deterministic values. This implies that 2f+1 replicas must have prepared m with n and nd. During a view change, the mechanism ensures that the new primary in view w must have collected the prepare records for m and, thus, the association of m with n and nd will be propagated from view v to view w. Hence, R_j cannot commit m to another n or nd.

If the replica non-determinism associated with m is of the type `VPOST` or `NPOST`, there exist only two cases: R_i is the primary in view v or R_i is a backup. If R_i is the primary (and it is not faulty), it can deliver m before the post-commit phase. However, this does not pose a problem because the Byzantine agreement for the set of non-deterministic values associated with m (recorded at R_i) is guaranteed to complete in view v. We have already proved the correctness of our mechanism in this case. Next, we consider the case where R_i is a backup. It delivers the message m only after it has reached Byzantine agreement for the ordering of both m and the set nd of non-deterministic values. Therefore, the association of m with n and nd must have been propagated from view v to view w, and R_j can commit m only with n and nd in view w. This

completes the proof.

6 Implementation and Performance Evaluation

We have implemented the mechanisms described in the previous section in C++ and integrated them into the BFT framework [6, 8, 9, 10]. The development and test platform consists of 14 HP blade servers running Ubuntu Server 9. Each of the blade servers is equipped with two Quad-Core Intel Xeon 2GHz CPUs with 5GB of memory. The nodes are connected via a Cisco Catalyst Blade Switch 3020 that offers full duplex 1 Gbps Ethernet connections.

The experiments described below focus on the evaluation of the overhead of providing Byzantine fault tolerance to the non-deterministic applications in the BFT layer. The cost associated with recording non-deterministic values, verifying those values, and replaying those values in the application layer is not studied in this work. First, we present the performance evaluation results using a single client with respect to various types of non-determinism and various size non-deterministic data (by non-deterministic data we mean the set of non-deterministic values associated with a type of non-determinism). Next, we present the results using various numbers of concurrent clients. Finally, we report the impact of our mechanisms on the end-to-end latency during view changes.

6.1 Basic Performance Evaluation

Figure 10 and Figure 11 show a summary of the end-to-end latency and throughput measurements for a client-server application under normal operation for different types of non-determinism. To avoid clutter, we have separated the results for single types of non-determinism from those for composite types of non-determinism; the results for single types are shown in the left figure, while those for composite types are shown in the right figure. In each iteration, each client issues a request to the server replicas and waits for the corresponding reply. There is no wait time between consecutive iterations. The size of each request and reply is fixed at 1 KByte. For each run, we measured the total elapsed time for 100,000 consecutive iterations

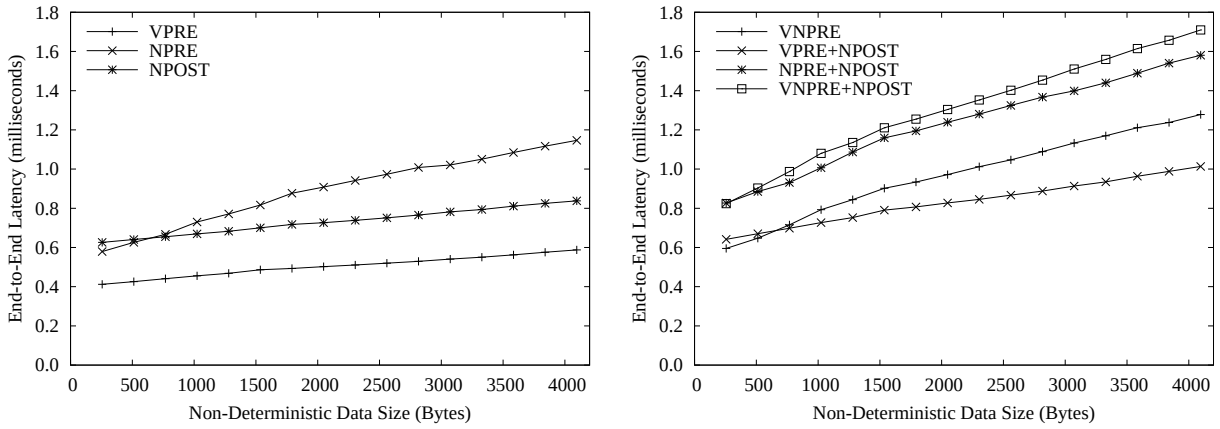

Figure 10: End-to-end latency for requests with different types of replica non-determinism under normal operation.

at each client, and calculated the average end-to-end latency and throughput.

The type of non-determinism and the size of non-deterministic data vary in different experiments, except for the throughput measurements, where the non-deterministic data are fixed at 256 Bytes for each type. Note that the sizes of non-deterministic data shown in Figure 10 on the horizontal axis are for *each* type, which means that, for composite types, the total size of non-deterministic data is twice or three-times as large as those displayed.

Except for the `VPRE`type, the handling of other types of non-determinism involves one or more phases of message exchange for non-faulty replicas to reach Byzantine agreement on the non-deterministic data. Thus, as shown in Figure 10, the end-to-end latency is noticeably larger, and the throughput is smaller, than for the `VPRE` type. The end-to-end latency difference is more significant as the size of non-deterministic data involved in each operation increases.

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 smaller, except for the `VPRE` type. These optimizations are described below.

In the pre-prepare-update phase, which is needed to handle `NPRE` non-determinism, each backup replica

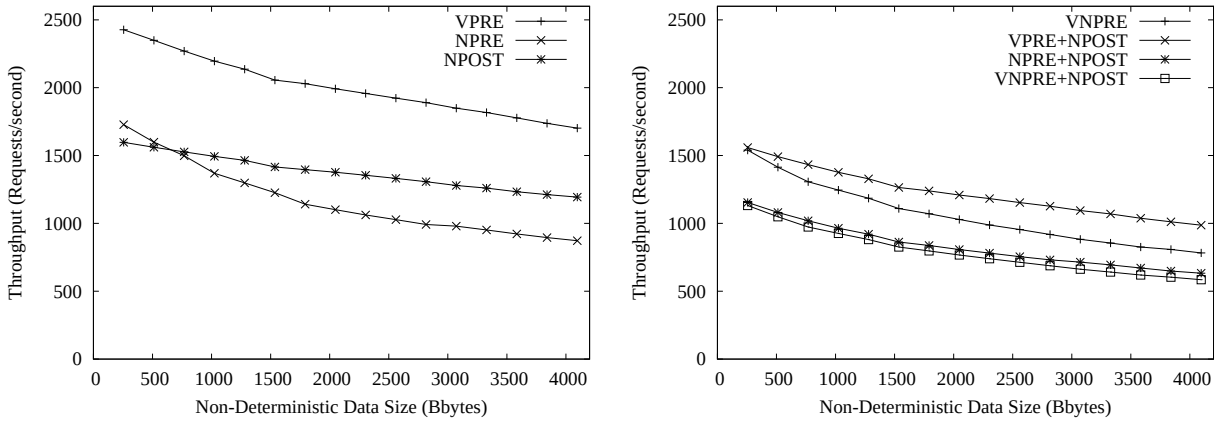

Figure 11: Throughput for requests with different types of replica non-determinism under normal operation.

multicasts its contribution of the non-deterministic data to all of the other replicas, and the primary decides on the collection (which includes the contributions from 2f+1 replicas, including its own) to be used to calculate the final non-deterministic data. Instead of multicasting the collection of non-deterministic data, the primary disseminates the collection of *digests* of the values proposed by each replica. This sharply reduces the message size if the size of non-deterministic data is large. Because each replica can log the non-deterministic data received from other replicas, a backup replica can verify the digests provided by the primary using its local copies. If a backup replica has not received the values proposed by one or more replicas included in the primary's message, 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. In this way, the Byzantine agreement for the non-deterministic data is reached together with that for ordering of the request, which reduces the number of messages needed to handle this type of non-determinism. Even though the end-to-end latency for a request increases slightly as a result, the system 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 non-deterministic data in conjunction with a null request so that the existing mechanisms can be reused.

It might be surprising to see in Figure 10 a crossover point of the graphs for the end-to-end latency for requests with NPRE non-determinism data and those with NPOST non-determinism data. When the size of the non-determinism data is small, the end-to-end latency for requests with NPRE non-determinism is smaller than that for requests with NPOST non-determinism. However, as the size of the non-determinism data increases, the latency for requests with NPRE non-determinism data increases rapidly and becomes greater than that for requests with NPOST non-determinism data. The reason is that, even with the optimization, the pre-prepare-update phase (needed to handle the NPRE type) still involves at least two large messages (one message per backup replica for its proposed non-deterministic values), while the post-commit phase (needed to handle the NPOST type) involves only one large message (sent by the primary). For the NPOST non-determinism type, there are two more rounds of message exchange than for the NPRE non-determinism type, which leads to a larger end-to-end latency when the size of the non-deterministic data is small. However, as the size of the non-determinism data becomes larger, the transmission delay for the messages that contain the non-determinism data begins to dominate, which results in a much faster increase in the end-to-end latency for requests with the NPRE non-determinism data, and eventually surpasses that for requests with NPOST non-determinism data. The crossover for the throughput results shown in Figure 11 occurs for the same reason.

To illustrate the overhead of our mechanisms for controlling various types of non-determinism with respect to the original BFT algorithm, we show in Figure 12(a) the end-to-end latency of each invocation when there is no replica non-determinism, and when each of the seven types of non-determinism (with a data size of 256 Bytes for each type of non-determinism) is present in the system. As expected, the end-to-end latency for VPRE non-determinism data is only slightly higher than that when no non-determinism is present, and the end-to-end latency is significant higher in the presence of NPOST non-determinism data, because it requires another around of Byzantine agreement. Nevertheless, the end-to-end latency is the largest (and the throughput is the smallest) in the presence of triple types of non-determinism (*i.e.,* VNPRE+NPOST).

The overhead of handling composite types of non-determinism is expected to be the sum of the overhead of handling each single type of non-determinism. To verify this, we computed the expected latency for each composite type by superimposing the overhead of each individual type of non-determinism involved, and

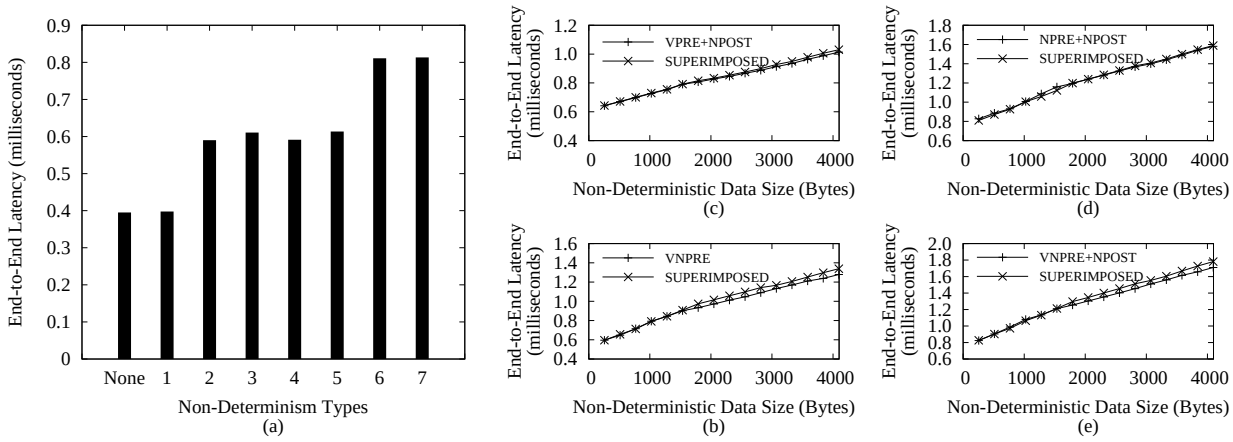

Figure 12: (a) End-to-end latency comparison without non-determinism, and with seven types of non-determinism (VPRE, NPRE, NPOST, VNPRE, VPRE+NPOST, NPRE+NPOST, VNPRE+NPOST) represented by the numbers from 1 through 7, respectively. (b)-(e) Comparisons for the measured latency and the expected value obtained by superimposing the overhead of individual non-determinism for the following composite types of non-determinism, (VNPRE, VPRE+NPOST, NPRE+NPOST, VNPRE+NPOST), respectively.

compared it with the measured latency. As shown in Figure 12(b)-(e), the computed latency is virtually identical to the measured result, confirming our expectation.

6.2 Performance Evaluation under Heavier Loads

To investigate the performance of our framework under heavier loads, we use multiple concurrent clients that issue requests at the replicated server. Each of the clients sends 100,000 request consecutively, where the size of the non-deterministic data is fixed at 256 Bytes. The results are summarized in Figure 13 and Figure 14. As for the basic performance evaluation, we present the results for the single types of non-determinism and those for the composite types of non-determinism separately.

Interestingly, the end-to-end latency results shown in Figure 13 also contains a crossover point, *i.e.,* the end-to-end latency for the requests with `NPOST` non-determinism data is greater than that for `NPRE` non-determinism data when the number of concurrent clients is less than 7. However, starting with 8 concurrent

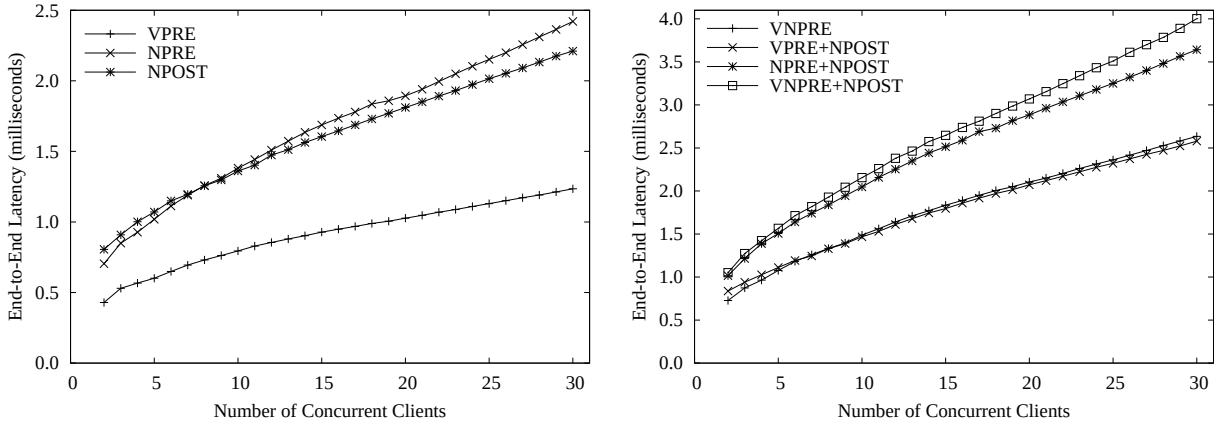

Figure 13: End-to-end latency for requests with different types of replica non-determinism in the presence of multiple concurrent clients under normal operation.

clients, the latency for the requests with `NPOST` non-determinism data falls below that of the requests with `NPRE` non-determinism data. The crossover can also be seen in the throughput measurement results in Figure 14, although it is less obvious.

The reason for this crossover is that, for requests with `NPOST` non-determinism data, when there is a sufficient number of concurrent clients, virtually all post-determinable non-deterministic data are piggybacked with the `pre-prepare` messages for other requests, rather than being sent as separate messages. The piggybacking mechanism effectively prevents the rapid increase in the end-to-end latency when the load on the system becomes higher and, similarly, helps to improve the throughput for requests with `NPOST` non-determinism data.

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

So far we have reported the experimental results for normal operation. In this section, we present the results that characterize the impact of our mechanisms on the performance of the Byzantine fault tolerance framework during view changes, *i.e.,* when the primary is faulty. We choose to use the end-to-end latency measured at the client as the metric to evaluate the impact of our mechanisms. In our experiment, a single client is used, and some requests generated by the client are instrumented, so that they trigger the crash of

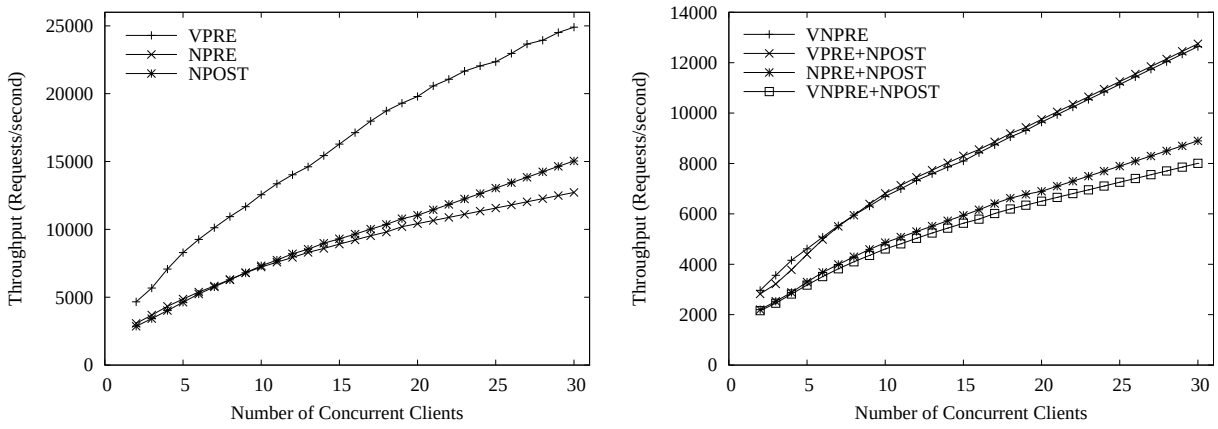

Figure 14: Throughput for operations with different types of replica non-determinism in the presence of multiple concurrent clients under normal operation.

the primary, which leads to a view change. Consequently, the end-to-end latency for such requests includes the round-trip latency during normal operation, the time it takes to detect the primary failure, and the view change latency. During the experiment, a 5-second view change timer, and a 150-millisecond message retransmission timer are used. Furthermore, a view change always succeeds and no message is lost during the view change.

The experimental results for various scenarios (different non-determinism types, including the scenario without any non-determinism, which is labeled *BFT with no ND*, and different sizes of non-deterministic data) are summarized in Table 1. As can be seen, the end-to-end latency remains virtually the same for all scenarios. This is expected because the handling of non-deterministic data during a view change has minimum impact on the view change latency. According to the modified view change mechanism described in Section 5.4, only the digest of the non-deterministic data is piggybacked onto the `view-change` and `new-view` messages. Therefore, the handling of non-determinism has virtually no effect on the performance of the view change. During our experiment, no message is lost. The view change latency might be larger if a message that contains non-determinism data is lost and has to be retransmitted.

Table 1: End-to-end latency during view changes.

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

7 Conclusion and Future Work

In this paper, we have presented the design and implementation of a Byzantine fault tolerance framework for non-deterministic applications. First, we described a novel classification of common types of application non-determinism based on two criteria: (1) whether or not the values associated with the non-deterministic operations can be determined prior to the execution of a request, and (2) whether or not the values proposed by one replica can be verified by another replica. This approach led to four types of non-determinism. Furthermore, we highlighted the risks incurred when such non-determinism is not controlled or not controlled properly by means of examples.

Based on the classification and risk analyses, we presented a set of mechanisms to control these types of non-determinism in a systematic and efficient manner. Our contributions here are three-fold. First, we observed that a heuristic bound must be used to verify a non-deterministic value (if it is verifiable), and argued that the safety property of the system is not violated if the bound is estimated incorrectly. Second, for non-verifiable pre-determinable operations, we noted that the collective inputs from 2f+1 replicas are required. Third, we illustrated a provisioning method that handles non-verifiable post-determinable operations so that the system can quickly recover replicas that have been damaged by malicious non-deterministic values sent by the primary. We also presented proofs of correctness for our mechanisms.

The implementation of these mechanisms is carried out by extending the well-known BFT framework

presented in [6, 8, 9, 10], which has very limited support for replica non-determinism. We have conducted extensive experiments to evaluate the performance of our prototype implementation. We have shown that our mechanisms incur only a moderate runtime overhead.

Our current implementation requires the application to provide a number of callback functions to identify and verify non-deterministic operations in the code for each remote method, and to record and replay such operations. It might require substantial expertise and time, on the part of the application developers, to analyze the application code and implement the callback functions correctly. To alleviate such a burden on the application developers, in future work, we plan to design tools that help analyze the source code for non-deterministic operations and that transparently record and replay non-deterministic operations.

The problem of having to deal with non-verifiable non-determinism is unique to the Byzantine fault model. Besides our current approach, we plan to explore an alternative solution, based on coin-tossing [11], that generates a common secret among the replicas. In this scheme, a threshold signature is used where each replica is dealt a share of a private key and the replicas collectively generate a group digital signature, which is later mapped to a common secret. The benefit of this approach is the reduced communication cost (no explicit Byzantine agreement on the common secret is necessary). However, the computation cost for generating a threshold signature might be significant. We aim to develop a guideline on when it is best to use which scheme, and eventually to add the capability to our frame such that the best scheme is selected dynamically based on the type of applications and runtime context.

Acknowledgment

We wish to thank the anonymous reviewers for their valuable comments and suggestions on an earlier version of this paper. This work was supported in part by NSF grant CNS 08-21319, and by a CSUSI grant from Cleveland State University.

References

[1] Arkin, B., Hill, F., Marks, S., Schmid, M., and Walls, T.: How we learned to cheat at online poker: A study in software security, http://www.developer.com/article.php/10915_616221_3/How-We-Learned-to-Cheat-at-Online-Poker-A-Study-in-Software-Security.htm, accessed August 2010.

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

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

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

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

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

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

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

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

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

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

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

[13] Dieter, W., and Lumpp, J.: User-level checkpointing for LinuxThreads programs, Proceedings of the USENIX Technical Conference, Boston, MA, June 2001, pp. 81–92.

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

[15] Lamport, L., Shostak, R., and Pease, M.: The Byzantine generals problem, ACM Transactions on Programming Languages and Systems, 1982, 4(3), pp. 382–401.

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

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

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

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

[20] Slember, J., and Narasimhan, P.: Living with non-determinism in replicated middleware applications, Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference, Melbourne, Australia, 2006, pages 81–100.

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

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

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

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

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

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