Byzantine Fault Tolerance for Nondeterministic Applications

Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University
2121 Euclid Ave, Cleveland, OH 44115
wenbing@ieee.org

Abstract

All practical applications contain some degree of nondeterminism. When such applications are replicated to achieve Byzantine fault tolerance (BFT), their nondeterministic operations must be sanitized to ensure replica consistency. To the best of our knowledge, only a specific type of nondeterminism, namely, the type whose values can be independently chosen by the primary and verified by other replicas prior to the execution of a request, has been successfully addressed under the Byzantine fault model. There are other types of nondeterminism in many practical applications. Some of them require the collaboration of all correct replicas, while others require a post-determination of a consistent set of nondeterministic values after the execution of a request at a particular replica. This paper points out the inadequacy of current approaches in handling such types of replica nondeterminism, and presents a systematic solution for the problem in the context of a unified BFT framework.

Keywords: Byzantine Fault Tolerance, Intrusion Tolerance, Security, Fault Tolerance Middleware

1. Introduction

Today's society has increasing reliance on services provided over the Internet. These services are expected to be highly dependable, which requires the applications that provide such services 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. The vulnerabilities due to insufficient design and poor implementation are often exploited by adversaries to cause a variety of damages, e.g., crashing of the applications, leaking of confidential information, modifying or deleting of critical data, or injecting of erroneous information into the application data. These malicious faults are often modeled as Byzantine faults. One approach to tackle such threats is to replicate the server-side applications and employ a Byzantine fault tolerance algorithm as described in [8, 9, 10, 11].

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, all practical applications contain some degree of nondeterminism. When such applications are replicated to achieve fault and intrusion tolerance, their nondeterministic operations must be sanitized to ensure replica consistency. To the best of our knowledge, only a specific type of nondeterminism, namely, the type whose values can be independently chosen by the primary replica and verified by other replicas *prior to* the execution of a client's request, has been successfully addressed in the past [8, 9, 10, 11]. The mechanisms designed to handle this type of nondeterminism either are not effective in guaranteeing replica consistency and/or are not effective in masking Byzantine fault, if the application to be replicated exhibits other types of nondeterministic behavior.

For example, in online poker games, such as Blackjack [1] and Texas Hold'em [18], pseudo-random number generators are used to shuffle the cards. The nature of the pseudo-random number generator is that if the seed remains the same, it will generate the same set of numbers in the same sequence. This means that once the seed is known, the order of the cards after shuffling, and consequently, the hands of each player, can be predicted deterministically. Therefore, the seed constitutes the most essential and confidential state of online poker applications. Considering the popularity of online poker games and the potential financial stakes involved, preventing cheating is urgent and a huge task for online gaming service providers. (Even though the cheating incidents are rarely reported.) As described in [1, 18], the primary cheating method is through estimating the seed by exploiting the weakness of the game server design, or through altering the server software with the help of collud-

ing insiders. Such threats can be countered by replicating the server application and running a Byzantine fault tolerance algorithm that ensures the use of a seed collectively contributed by all replicas for each shuffle operation. The traditional primary-based seed determination method does not work here because the backup replicas in general cannot verify the correctness of the seed proposed by the primary replica, which means that the primary can unilaterally decide on the seed value. (One might think that the time-of-day can be used as the seed, which the backup replicas *can* verify the value proposed by the primary. Unfortunately, this has been proved to be a bad idea because the seed can be easily estimated [18].) If the primary replica has been compromised, it may propose a seed that can be predicted by a colluding player.

Another type of nondeterminism is that caused by multithreading, which has become the more pervasive programming method used to build modern online applications. (This trend will only accelerate in light of the rapid development and deployment of multi-core processors.) The reason why the traditional method cannot handle this type of nondeterminism is that it is virtually impossible to predetermine the interleaving of threads *before* the execution of a request, considering the complexity and dynamic nature of the applications. Consequently, the primary replica is in no position to propose any thread ordering before it finishes processing a request, let alone for backup replicas to verify the correctness of such ordering.

In this paper, we introduce a classification of common types of replica nondeterminism present in many applications. We describe a set of mechanisms that can be used to sanitize these types of nondeterministic operations, and integrate them seamlessly with existing mechanisms in the seminal Byzantine fault tolerance (BFT) framework developed by Castro, Rodrigues, and Liskov [8, 9, 10, 11].

In summary, this paper makes the following research contributions:

- We provide two types of motivating applications to illustrate the inadequacy of current approaches to the problem of replica nondeterminism.

- We provide a classification of common types of replica nondeterminism useful both for Byzantine fault tolerance and benign fault tolerance.

- We propose a unified framework to ensure consistent Byzantine fault tolerant replication for applications exhibiting the nondeterministic behaviors 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 nondeterminism.

The remaining of the paper is organized as follows. Section 2 provides a brief introduction of the BFT technology developed by Castro, Rodrigues, and Liskov [8, 9, 10, 11] so that readers unfamiliar with this subject knows the context of the work described in this paper. Section 3 presents a classification of common types of replica nondeterminism. Section 4 presents a unified framework for Byzantine fault tolerant replication of nondeterministic applications. The focus is on our extensions to the original BFT algorithm. We also include in-depth discussions of two specific types of applications as potential target applications in applying our mechanisms and the associated security benefits. Section 5 describes our implementation and performance evaluation results. Section 6 provides an overview of related work. Finally, Section 7 summarizes this paper and points out future research directions.

2. Byzantine Fault Tolerance

This work is built on top of the BFT framework developed by Castro, Rodrigues, and Liskov [8, 9, 10, 11]. We use the same assumptions and system models as those of the BFT framework. For completeness, we briefly summarize the BFT framework here.

The BFT framework supports client-server applications running in an asynchronous distributed environment with a Byzantine fault model, *i.e.,* faulty nodes may exhibit arbitrary behavior. This requires the use of 3f+1 replicas to tolerate up to f faulty nodes. (In a recent publication [19], Yin et al. proposed a method to reduce the number of replicas to 2f+1 by separating the executing and agreement nodes.)

The BFT framework assumes that the replicas are single-threaded and all non-deterministic operations and their values are known *a priori* and verifiable by replicas prior to the execution of a request. The primary replica determines the values to be used for all replicas and disseminates them to the backup replicas The backup replicas subsequently verify the proposed values. If the primary replica is detected to be faulty, a view change is initiated. To prevent a faulty replica from intentionally initiating view changes, which is very expensive and may lead to denial-of-service attacks, the BFT algorithm starts a view change only if at least f+1 replicas have suspected the primary replica.

The BFT framework is implemented as a library to be linked to the application code (both the server and the client sides), as shown in Figure 1. In general, on the server side, we use the term *replica* to refer to the combined entity of the server application and the BFT library. On the client side, we use the term *client* to refer to the combined entity of the client application and the client-side BFT library.

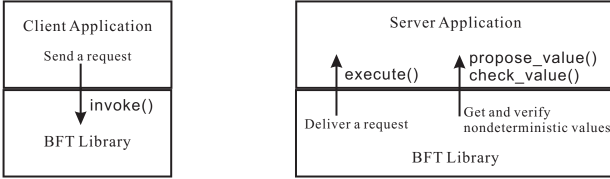

Figure 1. The positioning of the application and the BFT library, and the interfaces between the two components. Only the APIs directly related to this work are shown here.

Sometimes, however, it is necessary to distinguish the two parts explicitly. As shown in Figure 1, the client-server application and the BFT mechanisms (residing in the BFT library) interact via a set of Application Programming Interfaces (APIs). The APIs consist of a number of 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, for the client to send requests to the server replicas, for the server replicas to start the event loop managed by the BFT library, and for memory management. The APIs also consist of a number of upcalls to be implemented and supplied by the application, so that the BFT mechanisms can deliver a request to the server application, retrieve and verify nondeterministic values (if applicable), and retrieve and restore application state. Figure 1 includes a subset of the APIs directly related to this work.

In the BFT framework, a 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 on 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. To prevent a faulty primary replica from intentionally delaying a message, the client starts a timer after it sends out a request. It waits for f+1 identical replies from different replicas. Because at most f replicas are faulty, at least one reply must have come from a correct replica. If the timer expires before it receives a correct reply, the client broadcasts the request to all server replicas. This enables correct replicas to detect the primary failure so that a new primary can be elected (through a view change). All correct 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 replicas reliably in the same total order.

In the BFT framework, a three-phase algorithm, often referred to as the BFT algorithm, is used to ensure the total ordering of the requests received from different clients. The first phase is called the pre-prepare phase, where the primary replica multicasts a `PRE_PREPARE` message containing the ordering information, the client's request (or the digest of the request if it is large), and the nondeterministic values (if any) to all backup replicas. The backup replica then verifies the ordering information, the nondeterministic values, and the validity of the request message. If the backup replica accepts the `PRE_PREPARE` message, it multicasts to all other replicas a `PREPARE` message containing the ordering information and the digest of the request message being ordered. This starts the second phase, *i.e.,* the prepare phase. When a replica has collected at least 2f valid `PREPARE` messages for the request from other replicas, it multicasts a `COMMIT` message. This is the start of the third phase. When a replica has received at least 2f matching `COMMIT` messages from other replicas, the request message has been totally ordered and it is ready to be delivered to the server application. This concludes the third phase, *i.e.,* the commit phase, of the BFT algorithm. In the BFT framework, all messages are protected by a digital signature, or an authenticator [7] to ensure their integrity.

3. Classification of Replica Nondeterminism

We distinguish replica nondeterminism into the following three major categories:

- *Wrappable nondeterminism.* This type of replica nondeterminism can be easily sanitized 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. It is also possible to allow each replica to use its local ids. When such replica-dependent ids are to be propagated to external entities, they are translated to the group-wise values by a wrapper function (*i.e.,* there exists a deterministic mapping between the local values and the group-wise values). 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 [11].

- *Pre-determinable nondeterminism.* This is a type of replica nondeterminism whose values can be known *prior to* the execution of a client's request and it requires inter-replica coordination to ensure replica consistency.

- *Post-determinable nondeterminism.* This is a type of replica nondeterminism whose values can only be recorded *after* the request is submitted for execution

and the nondeterministic values won't be complete until the end of the execution. It also requires inter-replica coordination to ensure replica consistency.

In this paper, we will not have further discussion on the wrappable replica nondeterminism because it can be dealt with using a deterministic wrapper function without inter-replica coordination, and also because it has been thoroughly studied in [11]. Instead, we will focus on the rest of two types of replica nondeterminism that require inter-replica coordination.

Based on if a replica can verify the nondeterministic values proposed (or recorded) by another replica, replica nondeterminism can be further classified into the following types:

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

Overall, our classification gives four types of replica nondeterminism of our interest:

- *Verifiable pre-determinable nondeterminism.* This type of replica nondeterminism has be successfully dealt with in the original BFT framework [8]. We briefly summarize their mechanism as part of the unified framework in Section 4.1.
- *Non-verifiable pre-determinable nondeterminism.* This type of replica nondeterminism is discussed in this paper. If the application exhibits this type of nondeterminism, the replicas must collectively determine the nondeterministic values to prevent a single faulty replica from compromising the integrity of the service provided by the replicas. More discussion will follow in Section 4.2.
- *Verifiable post-determinable nondeterminism.* We have yet to identify a commonly used application that exhibits this type of nondeterminism. For completeness reason, we provide the mechanism needed to handle this type of nondeterminism as part of the unified framework in Section 4.3.
- *Non-verifiable post-determinable nondeterminism.* This type of replica nondeterminism is discussed in this paper. Ideally, the replicas should collectively determine the set of nondeterministic values to prevent a single faulty replica from compromising the health and integrity of other replicas. However, it is not clear if it is always feasible for the replicas to apply a deterministic algorithm to decide on a common set of values from those reported by individual replicas, as in the case of multithreading. 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 rely on the information reported by a single replica (*i.e.,* the primary replica) and to employ additional recovery mechanisms to minimize the impact of a faulty replica, as elaborated in Section 4.4.

4. A Unified Framework for Sanitizing Replica Nondeterminism

In this section, we present extensions to the BFT framework in handling all common types of replica nondeterminism. The unified framework requires intimate collaboration between the BFT mechanisms and the applications being replicated. Comparing with the APIs used in the BFT framework [11], the following server upcalls (*i.e.,* callback functions registered by the server application) are modified:

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

Where `seqno` is the sequence number assigned to the client's request under consideration, `req` is a pointer to the request message, `ndet_type` is a pointer to the type of the nondeterminism the replica might exhibit when executing the request, and `ndet` is a pointer to the buffer that stores the nondeterministic 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 nondeterminism (always the case) and, if applicable, the nondeterministic values received from the primary replica. 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 back to the caller in the return value.

The following upcall signature is not modified, but the interpretation of one of its parameters is changed:

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

The first parameter `req` is a pointer to the request message. The second parameter `rep` is a pointer to the reply message to be generated by the replica. The third parameter `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 an in-parameter. It is now reinterpreted as an in-out parameter. Depending on the type of replica nondeterminism, it might be an in-parameter as before, or, it might be an out-parameter when a replica has post-determinable nondeterminism and the function is invoked at the primary replica.

The following four types of replica nondeterminism are defined in the form of four constant integer values:

- `VERIFIABLE_PRE_DETERMINABLE`,
- `NONVERIFIABLE_PRE_DETERMINABLE`,
- `VERIFIABLE_POST_DETERMINABLE`,
- `NONVERIFIABLE_POST_DETERMINABLE`.

The constant names are self-explanatory.

The BFT algorithm is modified in the following ways. When a client's request arrives at the primary replica, if it is ready to order the message (when the number of ordered but not-yet executed messages is smaller than the window threshold), the primary replica invokes the `propose_value()` callback function registered by the application layer. The application supplies the type of nondeterminism that would be involved in the execution of the request, and if applicable, the nondeterministic values. Depending on the type of nondeterminism returned by the application, the modified BFT algorithm operates differently according to the mechanisms described from Section 4.1 through Section 4.4.

The modified BFT algorithm introduces two new types of control messages, namely, `PRE_PREPARE_UPDATE` and `POST_COMMIT`. The `PRE_PREPARE_UPDATE` message is used in the additional phase for the replicas to reach a Byzantine agreement on the collection of the nondeterministic values contributed by different replicas when nonverifiable pre-determinable nondeterminism is present. The `POST_COMMIT` message is used in the additional phase for the replicas to reach a Byzantine agreement on the nondeterministic values recorded by the primary replica after it has executed a request message (and hence, the name `POST_COMMIT`) when post-determinable nondeterminism is present.

4.1. Verifiable pre-determinable nondeterminism

If the nondeterminism for the operation at the primary replica is of type `VERIFIABLE_PRE_DETERMINABLE`, the application provides the nondeterministic values in the `ndet` parameter. The obtained information is included in the `PRE_PREPARE` message, and it is multicast to the backup replicas.

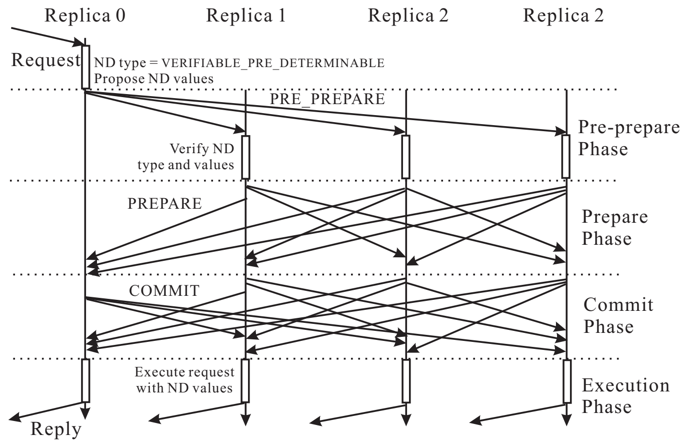

Figure 2. Normal operations of the modified BFT algorithm in handling verifiable pre-determinable nondeterminism. In the figure, ND is an acronym for nondeterminism, Replica 0 is the primary replica, and Replica 1 through 3 are the backup replicas.

On receiving the `PRE_PREPARE` message, a backup replica invokes the `check_value()` callback function. The replica passes the information received regarding the nondeterminism type and data values to the application layer so that the application can verify the following:

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

If either check turns out to be false, the `check_value()` call returns an error code, the backup replica then suspects the primary replica.

If it verifies the type of nondeterminism and the values proposed by the primary replica, and it accepts the client's request and the ordering information specified by the primary replica, the backup replica logs the `PRE_PREPARE` message and multicasts a `PREPARE` message to all other replicas. From now on, the algorithm works the same as that of the original BFT framework. The normal operations of the modified BFT algorithm is illustrated in Figure 2.

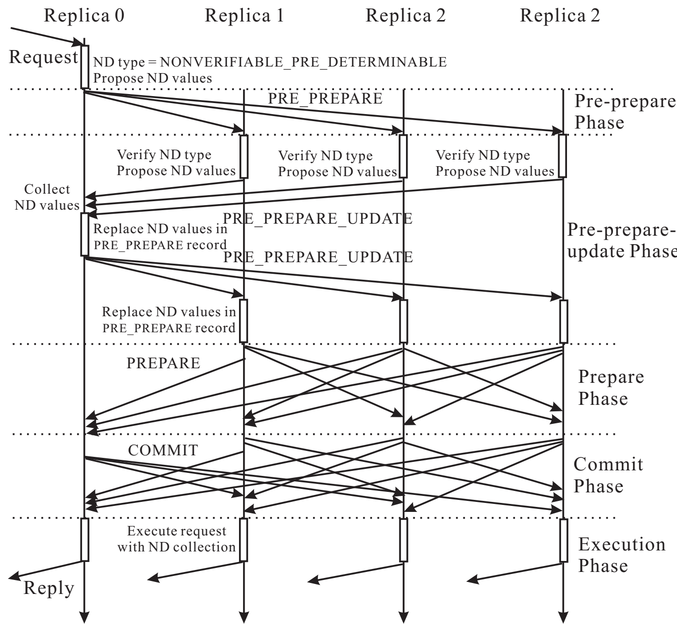

Figure 3. Normal operations of the modified BFT algorithm in handling non-verifiable pre-determinable nondeterminism.

4.2. Non-verifiable pre-determinable nondeterminism

If the nondeterminism for the operation at the primary replica is of type `NONVERIFIABLE_PRE_DETERMINABLE`, the application at the primary replica proposes its share of nondeterministic values in the `ndet` parameter. The type of nondeterminism and the nondeterministic values are included in the `PRE_PREPARE` message, and it is multicast to all backup replicas.

On receiving the `PRE_PREPARE` message, a backup replica checks the client's request and the ordering information as determined by the primary replica. If the `PRE_PREPARE` message passes the check, the backup replica subsequently invokes the `check_value()` callback function to verify the nondeterminism type information supplied by the primary replica. If the verification is successful, the backup replica invokes the `propose_value()` function to obtain its share of nondeterministic values. The backup replica then builds a `PRE_PREPARE_UPDATE` message including its own nondeterministic values, and sends the message to the primary replica.

When the primary replica receives at least 2f `PRE_PREPARE_UPDATE` messages from different backup replicas (for the same client's request), it builds a `PREPARE` message, including the 2f+1 sets of nondeterministic values, each protected by the proposer's digital signature or authenticator. The `PREPARE` message itself is further protected by the primary replica's signature or authenticator. The primary replica then multicasts the message to all backup replicas. From now on, the BFT algorithm operates according to the original algorithm, except that the 2f+1 sets of nondeterministic values are delivered to the application layer as part of the `execute()` upcall. Alternatively, the application could register a deterministic averaging function on the 2f+1 sets of nondeterministic values. A replica could invoke this callback function and delivers the average values to the application layer. The normal operations of the modified BFT algorithm for this type of nondeterminism is illustrated in Figure 3.

Having described the mechanism to be used to handle this type of replica nondeterminism, it is worthwhile to further discuss the type of applications that exhibit such nondeterminism and how our mechanism can be used to enhance the security and dependability of the applications.

For applications such as online porker games, the source of replica nondeterminism is often the most crucial state that should be protected because such values are used as the seeds to pseudo-random number generators for their operations, such as shuffling the cards. In fact, such applications rely on the randomness of the values to operate correctly. The process of retrieving such nondeterministic values is often called entropy gathering (in information theory, entropy is defined as a measure of uncertainty in the data collected). The values can be obtained either from a hardware device, such as a Geiger counter that counts the number of radioactive decays detected, or using a software solution, such as through sampling keyboard or mouse events in a computer [18]. On the other hand, if such values are not obtained from a high-entropy source, they may become predictable, and consequently, the system may be cheated [18].

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 et al.[19], to filter out 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 or covert channels to its colluding clients.

To counter such threats, such applications must be replicated using a Byzantine fault tolerant algorithm. Furthermore, each replica must use a different methodology to generate its nondeterministic values. In this case, a replica is in no position to verify the nondeterministic 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 ap-

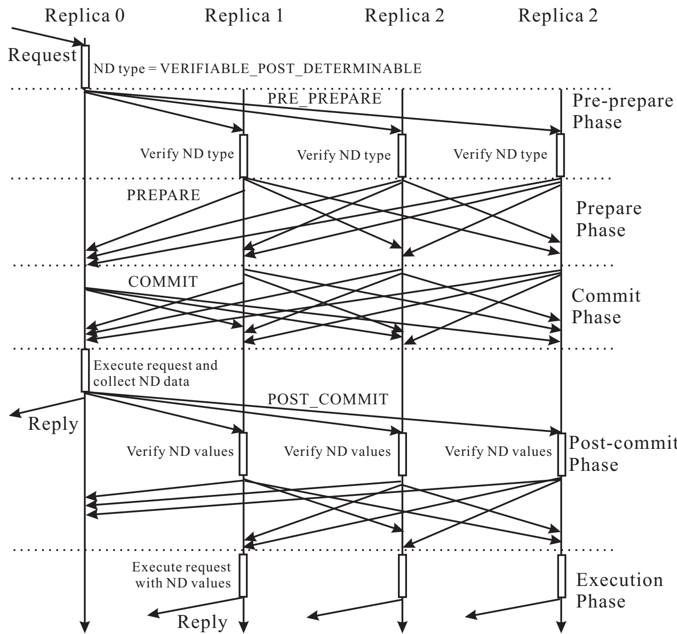

Figure 4. Normal operations of the modified BFT algorithm in handling verifiable post-determinable nondeterminism.

plying the mechanism described in this Section. This is very important, 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 defeat the purpose of providing Byzantine fault tolerance to applications.

4.3. Verifiable post-determinable nondeterminism

If the nondeterminism at the primary replica for the operation is of type `VERIFIABLE_POST_DETERMINABLE`, the application indicates the type in the `ndet_type` parameter and does not propose any nondeterministic values (it could not do so anyway per our definition). The primary replica includes the nondeterminism type information in the `PRE_PREPARE` message (without any nondeterministic 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 application at the backup replica confirms the type of nondeterminism associated with the client's request, the BFT algorithm proceeds to the commit phase as usual. Otherwise, the backup replica suspects the primary.

When the primary replica is ready to deliver the request message, it proceeds to performing the `execute()` upcall and expects to receive both the reply message (in the `rep` out-parameter) and the recorded nondeterministic values (in the `ndet` out-parameter). Once the `execute()` upcall returns, the primary replica builds a `POST_COMMIT` message containing the identity information for the client's request and the post-determined nondeterministic values. Then the primary replica multicasts the `POST_COMMIT` message to the backup replicas and sends the reply message to the client.

A backup replica cannot, however, deliver a request message immediately after the commit phase, instead, it must wait until the end of the post-commit phase. When it receives the `POST_COMMIT` message from the primary replica, a backup replica verifies the received nondeterministic values through the `check_value()` upcall. If the verification succeeds, the backup 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 it receives at least 2f `POST_COMMIT` messages with matching nondeterministic values from different replicas, a backup replica delivers the request message through the `execute()` upcall, together with the verified nondeterministic values. The backup replica then sends the reply to the client when the `execute()` call returns.

The normal operations of the modified BFT algorithm in handling this type of replica nondeterminism is summarized in Figure 4.

4.4. Non-verifiable post-determinable nondeterminism

The handling of non-verifiable post-determinable nondeterminism involves with the same steps as those described in the previous Section until the start of the post-commit phase.

The primary replica performs the `execute()` upcall and gets the reply and the nondeterministic values from the application. It sends the reply message to the client immediately. It then builds and multicasts a `POST_COMMIT` message with the 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 a backup replica receives the `POST_COMMIT` message and verifies the request information, it re-multicasts the message with its own signature or authenticator to all replicas. When it has collected at least 2f `POST_COMMIT` messages with matching non-deterministic values from other

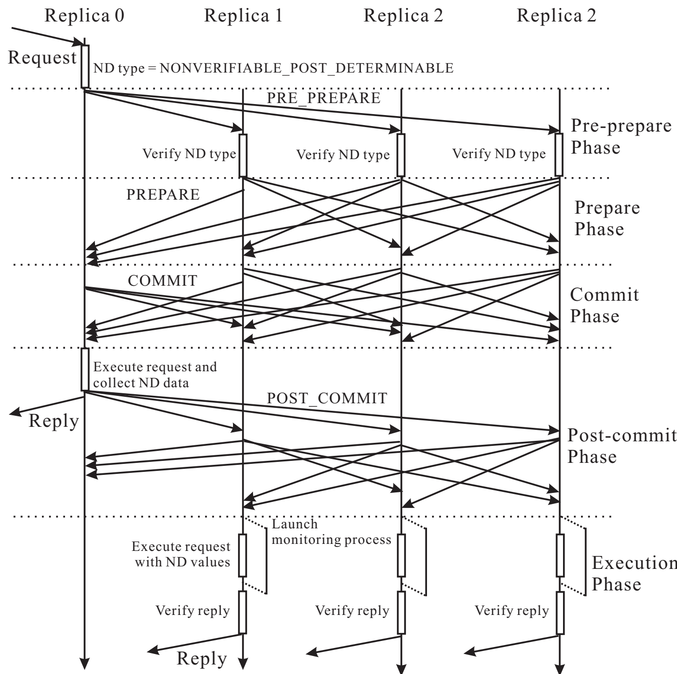

Figure 5. Normal operations of the modified BFT algorithm in handling non-verifiable post-determinable nondeterminism.

replicas, the backup replica prepares for the upcoming execution of the request message.

A faulty primary replica could disseminate a wrong set of nondeterministic values hoping to either confuse the backup replicas, or to block them from providing useful services to their clients. For example, if the nondeterministic data contains thread ordering information, a faulty primary replica can arrange the ordering in such a way that it may lead to the crash of the backup replicas (e.g., if the primary replica knows the existence of a software bug that leads to a segmentation fault), or it may cause a deadlock at the backup replicas (it is possible for a replica to perform a deadlock analysis before it follows the primary's ordering to prevent this from happening).

Because in general the replica cannot completely verify the correctness of the nondeterministic values until it actually executes the request, it is important for a backup replica to launch a separate monitoring process prior to invoking the `execute()` call. Should the replica run into a deadlock or a crash failure, the monitoring process can restart the replica and suspect the primary replica.

If the backup replica can successfully complete the `execute()` upcall, it compares the digest of its own reply message with that received from the primary replica. 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 if all correct backup replicas produce the same reply using the same set of nondeterministic values (even if they might be different from the set actually used by the primary replica, which implies that the primary replica is lying and will be suspected), the result is valid.

The normal operations of the modified BFT algorithm in handling this type of replica nondeterminism is shown in Figure 5.

A good example of this type of nondeterminism is that of multithreaded applications. When such applications are replicated, we must ensure different threads to access the shared data in the same order, otherwise, the state of different replicas may diverge. Due to the complexity and dynamic nature of multithreaded applications, it is virtually impossible to pre-impose an access ordering prior to 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 to other replicas so that they follow the same thread ordering, as described above.

4.5. Discussion

In practical applications, the execution of a request often involves with more than one type of nondeterminism, for example, both time-related nondeterminism (which is of verifiable pre-determinable type) and multithreading-related nondeterminism (which is of non-verifiable post-determinable type). To accommodate this complexity, a bitmask should be used instead of an integer value to capture the nondeterminism type information in the `propose_value()` and `check_value()` upcalls. However, the data structure used to store the nondeterministic values does not need to be made more sophisticated because it is the application's duty to generate and interpret them. The modified BFT algorithm can readily cope with this complexity as well. Using the same example, the time-related nondeterministic values can be determined during the pre-prepare-update phase. The multithreading-related nondeterminism can be resolved in the post-commit phase, as described in Section 4. The `ndet` parameter would contain both time-related and multithreading-related nondeterministic values when the `execute()` upcall is invoked by a backup replica.

5. Implementation and Performance Evaluation

We implemented the unified framework for nondeterministic applications by extending the BFT framework [8, 9, 10, 11] in C++. The current implementation is of proof-of-concept nature. There is significantly more work to be

carried out to improve the quality of the code and performance of the unified framework to be on par with those of the original BFT framework. Furthermore, we have yet to implement the facilities needed to capture and control the thread ordering on access to shared data. Consequently, 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.

The development and test platform consists of 5 personal computers each equipped with a Pentium 4 processor and 384 MB of RAM running RedHat 8.0 Linux. The computers are connected via a 100Mbps local area network. Figure 6 shows the summary of the end-to-end latency measurements for a simple client-server application under normal operations. The server is replicated on four of the computers and the client is running on the remaining computer. In each iteration, the 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 is fixed at 1KB. The type of nondeterminism and the size of nondeterministic values varies in different experiments. In each run, we measure the total elapsed time for 10,000 consecutive iterations. From the measured time, we derive the average end-to-end latency for each request-reply iteration.

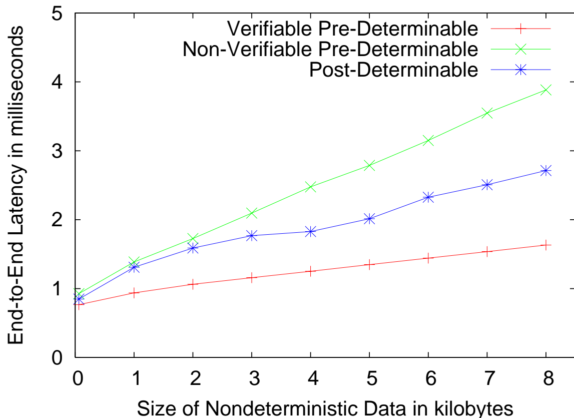

Figure 6. End-to-end latency for calls with different types of replica nondeterminism under normal operations.

As shown in Figure 6, we performed latency measurements for three different types of replica nondeterminism: verifiable pre-determinable, non-verifiable pre-determinable, and non-verifiable post-determinable nondeterminism. Since the cost for handling verifiable and non-verifiable post-determinable nondeterminism are similar under normal operations, we refer the measurement data for non-verifiable post-determinable nondeterminism simply as post-determinable in the figure and in the discussion below.

The handling of both non-verifiable pre-determinable and post-determinable nondeterminism involves with one additional phase of Byzantine agreement on the nondeterministic values. As such, the end-to-end latency is noticeably larger than that of verifiable pre-determinable nondeterministic operations, if there are large quantity of nondeterministic values. The results shown in Figure 6 are obtained after a number of optimizations to the mechanisms described in Section 4. Without these optimizations, the latency is significantly larger.

In the pre-prepare-update phase, which is needed to handle non-verifiable pre-determinable nondeterminism, each backup replica multicasts its contribution of the nondeterministic values to all other replicas, and the primary replica decides on the collection (must include the contributions from at least 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 replica disseminates the collection of the *digests* of the nondeterministic values from each replica. This sharply reduces the message size if the quantity of nondeterministic values is large. Since each replica can log the nondeterministic values received from other replicas, a (backup) replica can verify the digests provided by the primary replica with its local copies.

During the post-commit phase, which is needed to handle post-determinable nondeterminism, the primary replica disseminates the recorded nondeterministic values to all backup replicas, and each backup replica multicasts the digest of the received values instead of the values themselves to reduce the cost.

It is straightforward to understand why the cost of handling non-verifiable pre-determinable nondeterminism is much higher than that of handling post-determinable nondeterminism when there are large quantity of nondeterministic values. With the above optimization, the pre-prepare-update phase involves with at least 2 large messages (sent by the backup replicas) while the post-commit phase involves with only one large message (sent by the primary replica).

6. Related Work

Replica nondeterminism has been studied extensively under the benign fault model [2, 3, 4, 5, 6, 13, 14, 15, 16, 17, 20]. However, there is a lack of systematic classification of the common types of replica nondeterminism, and even less so on the unified handling of such nondeterminism.

[4, 5, 6, 17] did provide a classification of some types of replica nondeterminism. However, they largely fall within the types of wrappable nondeterminism and verifiable pre-determinable nondeterminism, with the exception of nondeterminism caused by asynchronous interrupts, which we do not address in this work.

The replica nondeterminism caused by multithreading has been studied separately from other types of nondeterminism, again, under the benign fault mode only, in [2, 3, 13, 14, 15, 16]. However, these studies provided valuable insight on how to approach the problem of ensuring consistent replication of multithreaded applications. It is realized that what matters in achieving replica consistency is to control the ordering of different threads on access of shared data. The mechanisms to record and to replay such ordering have been developed. So do those for checkpointing and restoring the state of multithreaded applications (for example, [12]). Even though these mechanisms alone are not sufficient to achieve Byzantine fault tolerance for multithreaded applications, they can be adapted and used for this goal. This work is an attempt to integrate such mechanisms with the BFT algorithm to render multithreaded applications Byzantine fault tolerant.

Under the Byzantine fault model, the main effort on the subject of replica nondeterminism handling so far is to cope with wrappable and verifiable pre-determinable replica nondeterminism [8, 9, 10, 11], In [8], Castro and Liskov provided a brief guideline on how to deal with the type of nondeterminism that requires collective determination of the nondeterministic values. The guideline is very important and useful, as we have followed in this work. However, we made substantial new contributions on top of their work. First, we made a strong case for the need of collective determination of nondeterministic values and its benefits for a type of practical applications. Second, we provided significantly more in-depth discussions and mechanisms on how to handle such type of nondeterminism. Third, we further considered the handling of multithreading-related replica nondeterminism, which has not been addressed in current literature under the Byzantine fault model. Finally, we also provided a classification of common types of replica nondeterminism and proposed a unified Byzantine fault tolerant framework to handle these types of nondeterminism.

7. Conclusion and Future Work

In this paper, we provided two types of motivating applications, namely, online poker applications and multithreaded applications, to illustrate the need for better mechanisms to handle the nondeterministic behaviors in such applications. We showed that if the traditional method is used, a compromised primary replica can easily break the Byzantine fault tolerance algorithm and destroy the integrity of the replicated service. Rather than providing ad-hoc solutions to the replica nondeterminism problems, we chose to follow a systematic approach. To do so, we first classified all common types of replica nondeterminism. Then, we presented the mechanisms needed to handle these types of replica nondeterminism under the context of a unified BFT framework for nondeterministic applications. Furthermore, we provided a proof-of-concept implementation of our mechanisms by extending the BFT framework [8, 9, 10, 11].

Our current implementation of the unified BFT framework is still in its early stage. We envisage that there are a lot of opportunities for us to optimize our implementation. For example, by piggybacking nondeterministic values with regular BFT messages, we may be able to eliminate the additional phases we introduced in some cases. We also plan to implement tools to help applications record, verify (if applicable), and replay nondeterministic values.

Acknowledgement

The author wishes to thank Professor Lorenzo Alvisi and his students for providing a bug-fix version of the BFT framework source code, and his help in understanding the BFT algorithm.

This work is sponsored by Cleveland State University through a Faculty Research Development award.

References

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

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

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

[4] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. And. Fault tolerance under Unix. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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