Testing the ”Fast Byzantine Consensus” Protocol

Alexandra Căruțașu[1]

Supervisors: Dr. Burcu Külahçıoğlu Özkan[1]**, João Miguel Louro Neto**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 26, 2025

Name of the student: Alexandra Căruțașu
Final project course: CSE3000 Research Project
Thesis committee: Dr. Burcu Kulahcioglu Ozkan, João Miguel Louro Neto, Dr. Jérémie Decouchant

Abstract

Byzantine fault-tolerant protocols have been around for decades, offering the guarantee of agreement on a correct value even in the presence of arbitrary failures. These protocols have become a critical part of achieving consensus in distributed systems and are widely used nowadays. As such, we should aim to ensure the correct functioning of these systems and one essential step to take in this direction is by finding systematic and automatic ways to test BFT protocols.

This paper evaluates the performance of ByzzFuzz, an automatic testing framework designed to find bugs in the implementation of Byzantine fault-tolerant protocols through randomized testing. In that sense, we evaluate ByzzFuzz's ability to find bugs in our implementation, compare its method of injecting network and process faults to a baseline method that arbitrarily injects faults and compare the performance in bug detection of small-scope and any-scope message mutations.

We implemented the "Fast Byzantine Consensus" protocol and employed ByzzFuzz to evaluate the framework's capability of finding implementation bugs. We materialized a liveness violation previously uncovered in a theoretical analysis research.

Keywords: Software testing, Byzantine fault tolerance, BFT protocols, Distributed algorithms, Fuzz testing

1 Introduction

Byzantine fault-tolerant protocols (BFT) were first formally described in 1982 by Lamport et al. in their seminal work on the Byzantine Generals Problem [1]. They are a class of algorithms designed to ensure the reliability and correctness of computing systems, particularly in distributed networks where components may fail or act maliciously. These protocols are named after the Byzantine Generals Problem, a theoretical scenario that illustrates the challenges of achieving consensus in a system where some participants might be unreliable or deceptive.

In a BFT system, multiple nodes (or 'generals') must agree on a single course of action (or 'consensus'), even if some nodes are faulty or compromised. The goal of BFT protocols is to ensure that the network continues to function correctly and make agreed-upon decisions despite these failures. This is crucial in environments like financial systems, cloud infrastructures, and blockchain technologies, where reliability and security are imperative.

The robustness of BFT protocols allows a system to handle a specific number of faulty nodes without hindering its ability to reach a correct consensus. These protocols typically involve complex mechanisms for proposing, validating, and agreeing on the state of the system through a series of messages exchanged between nodes. The ability to tolerate faults not only covers outright failures but also covers any form of deviation from the protocol, including malicious attacks.

In summary, BFT protocols ensure that correct nodes in a network will reach a consistent state and agree on the correct value as long as the number of arbitrary faulty nodes does not exceed the tolerance threshold, f. Over the years, these protocols have become essential in achieving consensus in distributed systems and the backbone of numerous applications, such as blockchain networks, distributed databases, and Internet-of-Things (IoT) systems.

Even though Byzantine fault-tolerant protocols are theoretically robust, providing strong guarantees and proofs on the correctness of the algorithms, translating them to code often reveals protocol mistakes or implementation errors. Errors in these protocols could potentially trigger critical situations affecting systems and people, such as allowing double spending in a cryptocurrency system, an adversary controlling additional nodes to disrupt the system, or overwriting valid data during resynchronization.

Because of the lack of enough automated and systematic ways to test Byzantine fault-tolerant protocols, violations in these protocols often go undetected and are only discovered years after their initial publication. This would be the case for the two failures discovered through theoretical analysis by Abraham et al. [2]: the liveness violation performed on the Fast Byzantine Consensus protocol [3] and the safety violation of the Zyzzyva protocol [4], 11 and 10 years respectively after their theoretical papers have been published. Three years after Abraham et al.'s paper was published, Bano et al. introduced Twins [5], an automated testing framework that could manually reproduce the bugs discussed in the theoretical analysis. Although Twins can materialize these bugs and it is shown that this is the case in the paper, there are no implementations or artifacts of the tool automatically reproducing these bugs. This highlights the critical importance of finding reliable, automated ways to test Byzantine fault-tolerant protocols to help identify potential issues. Additionally, other bugs have been revealed through theoretical analysis for Tendermint [6], PBFT [7], and Ripple [8], but not found yet through automated testing methods. Only ByzzFuzz [9] recently automatically uncovered a bug in the XRP Ledger consensus protocol of Ripple. This further motivates the demand for automated testing methods of BFT protocols.

The objective of this research is to address the need to develop automated, systematic ways to test Byzantine fault-tolerant protocols. More specifically, we evaluate the efficiency of the new testing framework developed by Winter et. al, ByzzFuzz [9], in finding bugs in our implementation of the Fast Byzantine Consensus protocol. ByzzFuzz is designed to automatically detect errors in BFT algorithms by injecting process and network faults during randomized test executions. The method uses fault-bounded and round-bounded testing for injecting faults during the execution and small-scope, structure-aware message mutations. Using this method, ByzzFuzz has previously effectively uncovered implementation bugs in consensus protocols.

We devised the following three research questions (RQ) to help us achieve this objective:

RQ1. Can ByzzFuzz find any bugs in the implementation of the Fast Byzantine Consensus protocol?

RQ2. How does the bug detection performance of ByzzFuzz compare to a baseline testing method that arbitrarily injects network and process faults?
RQ3. How do small-scope and any-scope message mutations of ByzzFuzz compare in their performance of bug detection for the selected protocol?

We implemented the Fast Byzantine Consensus protocol, "the first protocol that reaches asynchronous Byzantine consensus in two communication steps in the common case" [3], and employed the ByzzFuzz framework to test it.

Using ByzzFuzz, we uncovered both agreement and liveness violations in the implementation of our protocol, whose root cause stands in our adaptation of the protocol, not in the original protocol specifications. We then compared ByzzFuzz's fault-injection methods to a baseline testing method that arbitrarily injects process and network faults. Lastly, to evaluate ByzzFuzz's novel idea of introducing small-scope mutations, we assessed how small-scope mutations and any-scope mutations compare while testing the protocol.

The rest of the paper has the following structure: **Section 2** details the methodology employed in this research to evaluate the ByzzFuzz testing framework. **Section 3** dives into existing state-of-the-art solutions for testing Byzantine fault-tolerant protocols. In **Section 4** we explore our approach in implementing the Fast Byzantine Consensus protocol. **Section 5** describes the experimental setup used to uncover potential vulnerabilities in the protocol using ByzzFuzz, presents the results, and examines the implications of our findings. **Section 6** addresses the ethical considerations of our research and the reproducibility of our methods. **Section 7** discusses the results and their broader impact. Finally, **Section 8** concludes with a summary of our findings and outlines directions for future research.

2 Methodology

We address **Research Question 1** by testing our implementation of the Fast Byzantine Consensus protocol using ByzzFuzz.

In a test run, ByzzFuzz introduces a number of **process faults**, representing a period during the execution of the protocol during which a node may act maliciously, and **network faults**, when the network is not functioning properly and messages cannot be delivered as normally.

We conducted tests using various configurations of process and network faults during the protocol's execution to ensure a thorough evaluation of our implementation. This approach addresses the likelihood of encountering multiple network errors and the presence of malicious actors persistently attempting to disrupt the protocol across several rounds.

For each test scenario, we predefined the number of process and network faults, and executed the protocol multiple times. We then reported any abrupt termination, either caused by liveness and agreement violations or halted by error, and analyzed the root cause of the termination.

In **Research Question 2**, we compare the performance of ByzzFuzz to the baseline method that arbitrarily injects network and process faults. We replicated the methodology employed in **RQ1** to test our implementation of the Fast Byzantine Consensus under the arbitrary condition, which ensured that the comparison was evaluated under fair conditions.

The evaluation criteria we chose to measure how the two testing methods compare are the **number of unique violations** detected by the testing method and the **bug detection rate**.

ByzzFuzz employs more sophisticated approaches to identify bugs and, as such, we expect the baseline testing method to trigger less faulty scenarios since it lacks the precision in targeting the specific behaviors and rounds critical to BFT protocols. In order to confirm that ByzzFuzz's approach of injecting faults is more efficient than the baseline's method, we compared the **number of violations** detected by each testing approach. The **bug detection rate**, defined as the percentage of scenarios that terminate prematurely out of the total scenarios executed, is closely linked to the number of bugs identified during testing. We expected a higher evaluation score for ByzzFuzz, given that its methodology is likely to reveal more bugs by the conclusion of the testing process.

Another possible comparison criterion to assess the performance of ByzzFuzz as opposed to the baseline method was **the time to bug detection**, which could be defined through the number of events the protocol executes until it runs into a bug or physical time. As speed alone does not reflect how reliable the testing method is at finding bugs in a protocol, we set this criterion aside.

We reported the results, including which testing method was more efficient according to the evaluation criteria and what conclusions we can draw from this.

For **Research Question 3** we examine how different types of message mutations compare. We differentiate between **small-scope mutations** and **any-scope mutations**. Small-scope mutations involve minor modifications to the existing content of a message, such as incrementing or decrementing a number in the message. Conversely, any-scope mutations allow for more extensive changes. Our objective is to determine which mutation strategy is more effective at uncovering defects within the protocol.

To explore this question, we conducted a series of test runs, first running ByzzFuzz applying only small-scope, and then only any-scope mutations to protocol messages. We then assessed and compared the efficacy of each mutation type in identifying bugs, using metrics similar to those employed in **RQ2**—specifically, the **number of unique bugs** detected and the **bug detection rate**.

3 Related Work

Recent advancements in Byzantine fault-tolerant protocols have enhanced the robustness and performance of distributed systems, notably in blockchain technologies. However, testing the correctness and robustness of BFT protocols still poses a significant challenge, due to the complexity and non-determinism aspects of such protocols, which require cutting-edge solutions to create a robust testing methodology that is able to catch potential violations in scenarios.

In "Dissecting Tendermint" [6], Ammoussou-Guenou et al. deep-dive into the Tendermint protocol, particularly popular

for its use for consensus in blockchain systems. With this paper, vulnerabilities in older versions of Tendermint were identified through rigorous analysis of the protocol, related to its safety and liveness properties. While this paper does not explicitly provide an automated testing framework, it highlights the need for more rigorous testing frameworks that are able to reveal such vulnerabilities.

In a similar manner, Berger and Reiser [7] addressed the efficiency and correctness of the read operations within BFT protocols through theoretical analysis. They identified a critical flaw in the PBFT protocol's read-only optimization that could lead to liveness violations.

When it comes to the development of testing algorithms for Byzantine fault-tolerant protocols, Decouchant et al. [10] in their paper, "Liveness Checking of the HotStuff Protocol Family", extended the advancement by examining the liveness properties of the HotStuff protocol family. The authors utilized novel methods like temperature and lasso detection to identify scenarios where the protocol might fail the liveness property, showing that even well-designed protocols can suffer from subtle bugs that can impact the well-functioning of the protocols.

Dragoi et al. [11] introduce another novel approach in testing BFT protocols in their tool, Netrix. The authors leverage a domain-specific language (DSL) to enhance the testing efficiency for consensus implementation. This framework is unique as it combines networking infrastructure with DSL to control network events and guide the test process through programmer input. Netrix is able to identify inconsistencies and potential bugs by allowing the manipulation of network behaviors such as message delays, drops, and order modifications, which is essential in capturing rare but critical network faults.

Bano et al. [5] describes Twins, a cutting-edge testing framework that tests BFT protocols by duplicating nodes to simulate Byzantine faults. Each twin node in the system acts autonomously but shares the same identity and credentials with its twin, allowing it to emulate malicious behaviors. Twins' ability to replicate node behavior under fault conditions helps in identifying critical vulnerabilities that might not be evident through other testing methodologies, thus contributing to the advancements in providing more secure and reliable BFT systems.

Another innovative testing method for consensus implementations was introduced by Dragoi et al. in their seminal work, "Testing consensus implementations using communication closure" [12]. The author's approach was capitalizing on the concept of communication closure, an essential property in distributed systems that ensures that all messages that could influence the outcome of a computation have been communicated before progressing to the next step. Their method systematically leverages this property to enhance the realism of test scenarios, which are structured to mimic real-world conditions closely. By ensuring that every possible state transition is tested, including those that might occur under unexpected conditions, the approach can uncover bugs in consensus algorithms. This method also stresses the system under edge cases, providing a comprehensive testing suite that can effectively evaluate the system's resilience and fault tolerance.

LOKI [13], a state-aware fuzzing framework developed by Ma et al. advances the traditional fuzzing technique by adapting it to the changing states of blockchain consensus nodes. Unlike simple fuzzing methods that generate inputs blindly, LOKI monitors the states of the nodes and adjusts the test inputs accordingly, ensuring the tests remain relevant to the protocol execution. This innovative method of testing makes the testing framework effective at discovering memory-related vulnerabilities and logic-related bugs. The state-awareness of LOKI allows for more targeted testing, which is essential in complex systems where bugs may not manifest under normal conditions, but rather in more complex scenarios.

Together, these innovative approaches reflect significant advancements in the testing and validation of Byzantine Fault Tolerant systems, each contributing distinct methodologies and tools that enhance our understanding and capability to secure distributed consensus protocols.

4 Implementation of the "Fast Byzantine Consensus" protocol

In this section, we explore the implementation and adaptation of the Fast Byzantine Consensus protocol, focusing on its architecture and functionality. The protocol leverages ByzzBench, a framework developed by the Software Engineering research department of Delft University of Technology, to facilitate and benchmark Byzantine fault-tolerant protocol. Initially implemented as a single-shot consensus protocol, we further extended this protocol to accommodate multiple client requests through the introduction of both view and sequence numbers, enabling a multi-shot consensus process. This adaptation is essential for handling a series of client requests sequentially, ensuring each is processed before proceeding to the next. Additionally, we detail methods integrated into ByzzBench to detect liveness and agreement violations during the protocol's execution.

4.1 Introducing the "Fast Byzantine Consensus" protocol

One contribution of this research project is the implementation of the Fast Byzantine Consensus protocol. To implement the protocol, we followed the seminal work outlined by Martin et al. [3]. The paper provides the protocol architecture, including an overview of the messages, the methodologies for message handling and logic of the protocol operation method. We implemented the protocol on top of ByzzBench.

The protocol can be implemented in two versions: a *parametrized* version and a *non-parametrized* version. The *parametrized* version of the Fast Byzantine Consensus protocol allows for adjusting the number of processes based on the desired resilience to Byzantine failures and the need for fast consensus. Specifically, the parameterized version enables choosing different levels of fault tolerance (up to f faults) and operational speed (guaranteed 2-step operation up to t faults), by setting the total number of processes to $3f + 2t + 1$.

In contrast, the *non-parametrized* version of the protocol, consistently requires $5f + 1$ processes to operate. It is designed to always provide 2-step consensus in the common

case, irrespective of the number of Byzantine faults (up to f). This version prioritizes speed and simplicity in setup, always aiming for the minimum consensus steps.

In this paper, we focused on implementing and testing the non-parametrized version.

Particularities of the protocol included the assignment of roles to the nodes. A node can have one or more roles, **proposer**, **acceptor**, or **learner**, each having its specific responsibilities in achieving consensus during the protocol execution. We represented this as a list in our code, as outlined in line 2 in Algorithm 1.

In the common case, the protocol starts with a request from the client, which the leader processes. The leader then sends a proposed value to all acceptors in the network. If the acceptors have not previously committed to a value, they accept the new proposal and forward their acceptance to learners in the second step. Learners process the accepted values received from the acceptors and learn a value. Once they learn the value, they send the reply back to the client.

We implemented the handling of the messages as outlined in the paper. To have a way for the nodes to compute thresholds that some actions might require (for example, a learner learns a value once a quorum of acceptors accepted the same value), we introduced variables that hold the number of proposers, acceptors, learners and Byzantine nodes, as showed in line 3 under Algorithm 1.

The protocol poses as a single-shot consensus protocol, meaning that its functionality revolves around achieving consensus for one client request. This protocol does not differentiate between the *view number* and the *sequence number*, but rather uses only one variable, *proposal number*. The *proposal number* represents the leader at that moment and an acceptor will accept one proposal for a proposal number.

4.2 Adapting to multi-shot consensus protocol

As we were interested in how the protocol would work while handling multiple client requests, we extended the original protocol to handle multiple client requests. We introduce in the protocol the difference between a view number that is the equivalent of the proposal number as explained above, and a sequence number. To integrate the functionality of the sequence number in the protocol, we added a sequence number to all consensus-related messages and, whenever we check for the view number, we also check for the sequence number. The leader keeps track of the client requests and, if it receives a new request while there's another request being processed, we queue it until the current request is complete.

Figure 1 shows how the common case of the protocol executes with multiple client requests. The first client request will execute in 4 rounds of communication. The client sends the request to the proposers, the leader sends the PROPOSE message to acceptors, acceptors check the proposal and if they accept it, they send the ACCEPT message to the learners who, once they learn the value, they commit that value and send a reply back to the client. During this time, the client can send a new request to the leader. The leader will only start processing this new request once it has received confirmation that all learners have learned the value. That means that it will wait for all LEARN messages from the learners. After it receives the confirmation, it can progress to the next proposal.

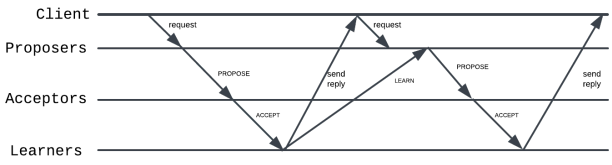

Figure 1: Common case execution with multiple requests

Lastly, lines 6-15 Algorithm in 1 are methods that handle all possible incoming messages received by the node, depending on the roles the replica has.

Algorithm 1 Pseudocode for FastByzantineReplica Class

```
1: Class FastByzantineReplica implements LeaderBased-
   ProtocolReplica
2: roles ← (list of roles attributed to the node)
3: int p, a, l, f (the number of proposers, acceptors, learners
   and faulty replicas in the system)
4: int proposalNumber
5: int sequenceNumber
6: handleProposeMessage : (sender, message)
7: handleAcceptMessage : (sender, message)
8: handleSatisfiedMessage : (sender, message)
9: handlePullMessage : (sender, message)
10: handleLearnMessage : (sender, message)
11: handleSuspectMessage : (sender, message)
12: handleQueryMessage : (sender, message)
13: handleReplyMessage : (sender, message)
14: handleViewChange : (sender, message)
15: handleNewViewChangeMessage : (sender, message)
```

4.3 Detecting Liveness and Agreement Violations

In our experimental setup, ByzzBench has the role of identifying Agreement and Liveness violations during the execution of the protocol. To automate the detection of these issues, ByzzBench employs specific algorithms, detailed in Algorithms 2 and 3, which define the conditions under which each type of violation is identified.

For Liveness violations, ByzzBench examines the state of message processing within each node. The condition for a Liveness Violation is that the protocol must never enter a state where all nodes are without any pending messages.

On the other hand, the Agreement predicate is designed to ensure all nodes commit the same values, which is crucial for the correct execution of consensus protocols. This check involves comparing the commit logs of all nodes; any discrepancy in these logs triggers an Agreement Violation. Such discrepancies could signify issues in state management or message handling, which could lead to incorrect or inconsistent outcomes across the distributed system.

By integrating these checks into the execution of our protocol scenarios, ByzzBench provides a robust mechanism for early detection of fundamental issues that could undermine the protocol's effectiveness and reliability.

Algorithm 2 Pseudocode for LivenessPredicate Class

```
1: Class LivenessPredicate implements ScenarioPredicate
2:
3: procedure TEST
4:     hasNoQueuedEvents← getTransport() .getEventsIn-
   State("QUEUED") .isEmpty()
5:     return ¬hasNoQueuedEvents
6: end procedure
```

Algorithm 3 Pseudocode for AgreementPredicate Class

```
1: Class AgreementPredicate implements ScenarioPredi-
   cate
2:
3: procedure TEST
4:     replicas ← Collect instances of Replica from execu-
   tor
5:     commonPrefixLength ← Max length of commit logs
   from replicas
6:     for i = 0 to commonPrefixLength - 1 do
7:         distinctIthEntries ← Collect distinct entries at in-
   dex i from each replica's commit log
8:         if size of distinctIthEntries > 1 then
9:             return false
10:        end if
11:    end for
12:    return true
13: end procedure
```

5 Experimental Setup and Results

This section outlines the experimental framework and findings from testing our implementation of the Fast Byzantine Consensus protocol using the ByzzFuzz framework. We initially focus on a non-parametrized version of the protocol, employing ByzzFuzz to explore the robustness of the protocol against various types and intensities of faults, and evaluate the framework's efficiency in uncovering bugs in BFT protocols.

Further experimentation involved running the protocol in its parametrized form using ByzzBench, specifically targeting known vulnerabilities as described in theoretical analyses. This involved materializing a specific fault scenario that prevents consensus from being achieved.

5.1 Testing the non-parametrized version with ByzzFuzz

The first step in testing our BFT protocol implementation is to set up the testing environment in ByzzBench. This setup will be used for both testing methodologies (ByzzFuzz and baseline) and is comprised of the following parameters:

- **number of scenarios** each test run executes for.
- **minimum events**, which checks if N events have been scheduled during a scenario and terminates the execution if the threshold has been reached. This builds on the heuristic that most bugs can be reproduced with a small number of events from the starting state.
- **minimum rounds**, which checks if N rounds have elapsed and terminates the execution.
- **sampling frequency**, which verifies every N scheduled events if protocol execution should terminate.

Table 1 provides a detailed overview of the parameter configurations. These settings were selected based on preliminary trials that suggested an optimal balance between the discovery of critical issues and computational expense.

Parameter	Value
Number of scenarios	3000
Minimum events	500
Minimum rounds	5
Sampling frequency	50

Table 1: Experiments Configuration

RQ1. We employed ByzzFuzz and tested our implementation of the "Fast Byzantine Consensus" protocol in the non-parametrized version with the experimental setup as shown in Table 1.

ByzzFuzz introduces the concept of network faults by injecting network partitions during a randomly selected message round, resulting in the dropping of all messages from different network partitions for that specific round of communication. In the case of process faults, the ByzzFuzz algorithm mutates the messages sent from a randomly chosen node to a subset of replicas within the network during a random message round. This mutation involves selecting a random small-scope alteration from Table 4.

Table 2 shows the results we collected after testing our implementation of the protocol in a network with 6 nodes, varying the number of *process* = [0,2] and *network* = [0,2] faults injected among $r = 10$ rounds of execution.

Table 2: ByzzFuzz and baseline execution results

Faults injected		Terminated By			
Process	**Network**	**L**	**D**	**Max Actions**	**Errors**
0	0	0	0	3000	0
1	0	3	17	2980	0
2	0	8	22	2970	0
0	1	139	0	2861	0
0	2	203	0	2797	0
1	1	137	7	2856	0
baseline	baseline	4	0	2996	0

Testing the implementation of our protocol using ByzzFuzz with $p = 1$ or 2 process faults led to the discovery of disagreement violations. By analyzing the faulty scenarios,

we traced back the cause to implementation omissions in our adaptation of the protocol for multi-shot operations, not flaws in the protocol's original design.

ByzzFuzz uses small-scope mutations to introduce small deviations in the original message content, which in our case can mistakenly lead the replicas to restart prematurely. This arises from an oversimplification in our implementation. Specifically, when a replica receives a consensus-related message (such as PROPOSE, ACCEPT, or LEARN) that includes a sequence number one higher than its current number, the replica immediately jumps to this next sequence number and restarts. In the scenarios that end in disagreement, ByzzFuzz introduces small-scope mutations in the sequence number of these messages and replicas prematurely advance, which causes some nodes to fail committing to one of the proposals like they should have, resulting in discrepancies between the commit logs of different replicas.

In Figure 2 we give an example of how such disagreement violations manifest. We omitted messages that are exchanged normally and don't affect the execution. Node F, who is an acceptor in the network, receives the first PROPOSE message with a sequence number higher by one than its initial sequence number. It incorrectly restarts and moves to the next sequence number, but as a learner, it learns and commits the value regardless. In the second proposal, since it does not move to the next sequence number because the new PROPOSE message contains the same sequence number as the replica, it skips committing, as it has already committed. In the third proposal, it commits as normal. In the end, learners C, D, and E in the diagram will have committed three times, while F two times, causing a discrepancy in the commit log.

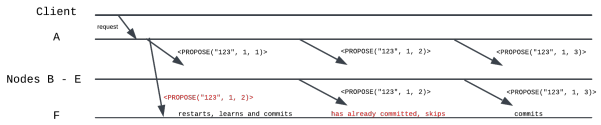

Figure 2: Example of disagreement bug

ByzzFuzz efficiently uncovers this oversimplification in our implementations, as we need more robust handling of sequence numbers and message processing within the protocol to prevent premature restarts and ensure consistency across all replicas' logs.

The liveness violations that manifest under process faults p = [0, 2] in Table 2 also stem from an oversimplified approach to transitioning between proposal rounds. These violations occur when the proposer does not advance to the next sequence number as expected. This is triggered when the proposer receives a mutated message with the sequence number higher by two, instead of one. Due to this unexpected sequence number, the proposer does not progress, leading to several issues that block the replica from progressing: not enough acceptors accept the proposed value, insufficient proposers suspect the current leader, and consequently, the proposed value fails to achieve consensus. This stagnation prevents the leader from advancing the process, ultimately causing the protocol to become stuck without making further progress.

The liveness violations that were reported while injecting n = [0,2] network faults using ByzzFuzz are "potential liveness issues", meaning they occur under network faults that violate the protocol assumptions. In the current implementation of ByzzBench, dropped messages are not buffered and returned back to the network. This implementation violates the protocol assumption which states that messages that are sent infinitely many times will eventually be delivered. Since some of the messages necessary to progress to the next round in our oversimplified implementation of the multi-shot consensus are dropped, replicas remain stuck in the previous proposal, making it impossible for the leader to make progress or for the leader election to take place.

For $p = 1$ and $n = 1$, the protocol runs into the same root causes that trigger the liveness and agreement violations when running with just process faults or network faults.

After running the experiments, we conclude that ByzzFuzz can uncover bugs in our implementation of the protocol. By exploiting small-scope mutations or introducing network partitions, ByzzFuzz triggered agreement and liveness violations that can legitimately arise during a faulty execution. These terminations are not caused by a mistake in the original protocol, but due to oversimplifications introduced by us in attempting to adapt this protocol to a multi-shot version.

RQ2. To answer this research question, we ran the baseline testing method that arbitrarily injects network and process faults during the execution of the protocol using the same experimental configuration as in table 1. The result of running the baseline testing on our implementation of the protocol is found in Table 2.

In Table 3, we present a comparative analysis of the faulty scenarios identified using ByzzFuzz versus those detected using the baseline testing algorithm. For comparison, we use the two evaluation metrics: (1) the number of scenarios that concluded with a termination condition, and (2) the detection rate, the proportion of faulty scenarios relative to the total number of scenarios executed, here fixed at 3000.

Table 3: ByzzFuzz and arbitrary fault injection comparison

Faults	No. of Violations	Detection rate
p = 1, n = 0	20	0.67%
p = 2, n = 0	30	1.00%
p = 0, n = 1	139	4.63%
p = 0, n = 2	203	6.77%
p = 1, n = 1	133	4.75%
baseline	4	0.13%

The liveness violations discovered by the baseline testing have the same root cause as the liveness violations discovered by ByzzFuzz in **RQ1**, the leader's inability to progress to the next round when it has not received enough answers from the learners. We did not observe any new liveness terminations triggered by the baseline method.

However, the baseline testing algorithm was unable to uncover agreement violations, confirming that ByzzFuzz's

method of introducing round-based small-scope mutations is more efficient at uncovering protocol bugs in our implementation of the "Fast Byzantine Consensus" protocol.

RQ3. To evaluate how small-scope mutations and any-scope mutations compare while testing our implementation of the protocol, we ran the ByzzFuzz framework again for 1000 scenarios under p = [1, 2] process faults injected in r = 10 rounds.

Table 4 presents the possible small-scope message mutations that ByzzFuzz can apply to the messages of "Fast Byzantine Consensus" protocol. In the message content, v represents the value sent in the message, p represents the proposal number (i.e. view number) and s represents the sequence number. For any-scope mutations, we apply the modifications to the same part of the message, but instead of applying small deviations to the message content, we replace the content with an arbitrary value.

Table 4: Small-scope message mutations

Message	Mutations
$\langle$PROPOSE, $(v, p, s), pc\rangle$	$\langle$PROPOSE, $(v, p, s{+}1), pc\rangle$ $\langle$PROPOSE, $(v, p, s{-}1), pc\rangle$ $\langle$PROPOSE, $(v', p, s), pc\rangle$
$\langle$ACCEPT, $(v, p, s)\rangle$	$\langle$ACCEPT, $(v, p, s{+}1)\rangle$ $\langle$ACCEPT, $(v, p, s{-}1)\rangle$ $\langle$ACCEPT, $(v, p{+}1, s)\rangle$
$\langle$SATISFIED, $(v, p, s)\rangle$	$\langle$SATISFIED, $(v, p{+}1, s)\rangle$ $\langle$SATISFIED, $(v, p{-}1, s)\rangle$
$\langle$LEARN, $(v, p, s)\rangle$	$\langle$LEARN, $(v, p, s{+}1)\rangle$ $\langle$LEARN, $(v, p, s{-}1)\rangle$
$\langle$PULL, $(v, p, s)\rangle$	$\langle$PULL, $(v, p, s{+}1)\rangle$ $\langle$PULL, $(v, p, s{-}1)\rangle$
$\langle$QUERY, $(v, p, s)\rangle$	$\langle$QUERY, $(v, p{+}1, s)\rangle$ $\langle$QUERY, $(v, p{-}1, s)\rangle$
$\langle$REPLY, $(v, p)\rangle$	$\langle$REPLY, $(v, p{+}1)\rangle$ $\langle$REPLY, $(v, p{-}1)\rangle$
$\langle$VIEW-CHANGE, $(p)\rangle$	$\langle$VIEW-CHANGE, $(p{+}1)\rangle$ $\langle$VIEW-CHANGE, $(p{-}1)\rangle$

In Table 5 we describe the results obtained when running ByzzFuzz using only small-scope mutations and any-scope mutations against p = [1, 2] process faults.

Table 5: Small-scope and any-scope comparison

	p	L	A	Detection rate
small-scope	1	1	7	0.8%
small-scope	2	3	8	1.1%
any-scope	1	5	0	0.5%
any-scope	2	5	0	0.5%

In our evaluation, the agreement violations discovered when running ByzzFuzz with small-scope mutations can not be uncovered by any-scope mutations. As explained in the results of **RQ1**, the disagreement in the replica's commit logs is caused by the oversimplification of our implementation, scenario which can be revealed only through small-scope mutations.

Testing with any-scope mutations reveals liveness violations in our implementation, which occur due to messages received with random sequence or view numbers that block the leader from progressing. This is the same root cause as with the liveness violations in small-scope testing. No new liveness violations were identified with any-scope mutations that did not also occur with small-scope mutations.

While testing our implementation of the "Fast Byzantine Consensus" protocol, small-scope mutations proved to be more efficient than any-scope mutations at uncovering flaws and potential faulty scenarios.

5.2 Running the parametrized version with ByzzBench

Abraham et. al uncovered in the theoretical analysis "Revisiting fast practical byzantine fault tolerance" a scenario under which the Fast Byzantine Consensus protocol is not able to progress and achieve consensus under the parametrized scenario [2]. We implemented the required non-parametrized set-up to run the protocol and materialized the bug manually in ByzzBench.

Figure 3 shows the communication steps that take place in order to achieve this state. We followed the protocol communication steps as explained in the theoretical analysis and materialized this scenario, confirming that the protocol achieves the same final state as in the protocol paper.

In the current implementation of ByzzBench and ByzzFuzz, even though ByzzFuzz is capable of generating the necessary faults to accurately replicate this specific scenario, the liveness algorithm employed by ByzzBench does not successfully identify this faulty execution. The cause for this is that the protocol continuously elects new leaders in an endless loop in the attempt to reach consensus. This endless cycle leads to new messages being constantly queued without resolution. Algorithm 2 in ByzzBench then fails to recognize the impossibility of reaching termination under these circumstances. Recognizing this limitation, we plan to address this in future updates of ByzzBench.

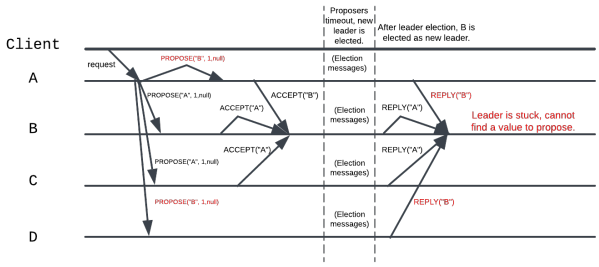

Figure 3: Materialized bug

6 Responsible Research

As part of our responsibility as researchers, we ensured that every aspect of the research process, including the implementation details of the protocol, the adaptations made to handle multiple client requests, and the testing environment setup, is thoroughly documented.

We detailed the metrics and criteria used for evaluating the protocol's performance and the effectiveness of the testing framework. This clarity helps other researchers understand and replicate our evaluation methods to verify findings or explore further.

Given the comprehensive overview provided of both the methodology and experimental procedures, we have taken the necessary steps to ensure that our research is reproducible. This detailed documentation supports the validity of our findings and enables other researchers to replicate and validate our results effectively.

In the future, we plan to publish and make the code publicly accessible, ensuring that the broader research community can utilize, evaluate, and build upon our work. This commitment to openness is intended to foster collaboration and accelerate innovation in the field. By sharing the code, we aim to enable the community to further replicate our findings, propose enhancements, and integrate new functionalities, thus enriching the collective knowledge base and advancing the state of the art in testing Byzantine fault-tolerant protocols.

Another highly important component of our research is the commitment to responsible disclosure. In our study, we found that the original protocol did not contain any inherent flaws, thus negating the need for disclosure. However, had we identified any significant issues, the proper course of action would have involved privately contacting the protocol's authors to inform them of the findings. This would allow them the opportunity to address and rectify any errors before public dissemination, ensuring a responsible approach to enhancing the security and reliability of the protocol.

7 Discussion

Our research focused on implementing the Fast Byzantine Consensus protocol, adapting it to handle multiple client requests and conducting extensive testing experiments to rigorously evaluate and compare different testing approaches.

The experimental outcomes demonstrated ByzzFuzz's efficiency in testing our implementation of the Fast Byzantine Consensus protocol. The implementation of round-based injection and small-scope mutations in the ByzzFuzz algorithm effectively uncovered flaws caused by our simplified approach at adapting this single-shot consensus protocol to multi-shot.

Our findings indicate that ByzzFuzz is proficient in detecting both agreement and liveness violations, which are essential for analyzing the reliability and security of Byzantine fault-tolerant systems. The framework's capability to effectively identify these violations under different fault scenarios showcases its potential as a robust tool for testing BFT protocols.

Comparing the performance of ByzzFuzz with the baseline testing method revealed that ByzzFuzz is more effective in detecting complex bugs. While both testing methods were capable of identifying liveness violations, ByzzFuzz was uniquely successful in uncovering the disagreement bug, which the baseline method did not detect. This indicates that ByzzFuzz's testing methodology is better at exploring faulty behaviors in the implementation of our protocol than the more generalized approach of the baseline method.

Lastly, through comparative analysis of the results obtained when running ByzzFuzz using only small-scope mutations versus only any-scope mutations, we found that only small-scope mutations uncovered scenarios that could end in disagreement, rendering them more efficient.

In the future, we want to improve our implementation of the Fast Byzantine Consensus protocol, particularly focusing on increasing its resilience during multi-shot executions. Our current simplified approach, while effective, lacks the complexity needed to robustly manage simultaneous multiple states. We plan to develop a more sophisticated mechanism that allows replicas to handle multiple states concurrently. This will involve designing a system where replicas can progress to subsequent rounds only after achieving a higher level of certainty regarding the correctness and safety of moving to the next proposal. By integrating this enhanced capability, we aim to significantly improve the protocol's efficiency and reliability in multiple client requests.

Another point of improvement would be to enhance ByzzBench's liveness detection system, which would add the necessary functionality to detect the faulty scenario in Abraham's et. al paper [2].

8 Conclusions

Our primary research objective was to assess the effectiveness of ByzzFuzz as a testing tool for Byzantine fault-tolerant protocols. We addressed this objective by answering the three research questions, which evaluated ByzzFuzz's ability to uncover bugs in our protocol implementation, compared its bug detection efficacy to a baseline method, and analyzed the performance of small-scope versus any-scope message mutations in bug identification.

In our study, we found that ByzzFuzz proved to be a highly effective tool for identifying bugs in Byzantine fault-tolerant protocols. Our results indicated that ByzzFuzz outperformed the baseline method in detecting bugs. Furthermore, the small-scope message mutations were particularly effective, uncovering a broader range of bugs compared to any-scope mutations. This suggests that ByzzFuzz's approach to testing and message mutation significantly enhances the robustness of protocol testing.

Lastly, using ByzzBench, we manually materialized the bug found in Abraham et al.'s paper, "Revisiting fast practical byzantine fault tolerance" [2].

The findings from this study highlight the importance of specialized testing frameworks like ByzzFuzz in the development of Byzantine Fault Tolerant protocols. By identifying critical vulnerabilities that could compromise consensus, ByzzFuzz helps in refining the protocol to withstand real-

world adversarial conditions.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, Jul. 1982. [Online]. Available: https://doi.org/10.1145/357172.357176

[2] I. Abraham, G. Golan-Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin, "Revisiting fast practical byzantine fault tolerance," *ArXiv*, vol. abs/1712.01367, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:7902429

[3] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, Jan. 2010. [Online]. Available: https://doi.org/10.1145/1658357.1658358

[5] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," in *International Conference on Principles of Distributed Systems*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:237235563

[6] Y. Amoussou-Guenou, A. Del Pozzo, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Dissecting tendermint," in *Networked Systems*, M. F. Atig and A. A. Schwarzmann, Eds. Cham: Springer International Publishing, 2019, pp. 166–182.

[7] C. Berger, H. P. Reiser, and A. Bessani, "Making reads in bft state machine replication fast, linearizable, and live," in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, 2021, pp. 1–12.

[8] I. Amores-Sesar, C. Cachin, and J. Mićić, "Security Analysis of Ripple Consensus," in *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), Q. Bramas, R. Oshman, and P. Romano, Eds., vol. 184. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 10:1–10:16. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2020.10

[9] L. Winter, F. Buse, D. Graaf, K. Gleissenthall, and B. Kulahcioglu Ozkan, "Randomized testing of byzantine fault tolerant algorithms," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 757–788, 04 2023.

[10] J. Decouchant, B. K. Ozkan, and Y. Zhou, "Liveness checking of the hotstuff protocol family," in *2023 IEEE 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2023, pp. 168–179.

[11] C. Dragoi, S. Nagendra, and M. Srivas, "A domain specific language for testing distributed protocol implementations," in *Networked Systems: 12th International Conference, NETYS 2024, Rabat, Morocco, May 29–31, 2024, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 100–117. [Online]. Available: https://doi.org/10.1007/978-3-031-67321-4_6

[12] C. Drăgoi, C. Enea, B. K. Ozkan, R. Majumdar, and F. Niksic, "Testing consensus implementations using communication closure," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428278

[13] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, and H. Li, "Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols," 01 2023.