Implementing and Preforming Randomized Tests on the HotStuff BFT Protocol

Lubomir Marinski

Supervisor: João Miguel Louro Neto

Responsible Professor: Dr. Burcu Kulahcioglu Özkan

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: Lubomir Marinski
Final project course: CSE3000 Research Project
Thesis committee: Dr. Burcu Kulahcioglu Özkan, João Miguel Louro Neto, Dr. Jérémie Decouchant

Abstract

Although Byzantine Fault Tolerant (BFT) protocols such as HotStuff are nominally resistant to a number of faulty or unreliable participants, implementation or design errors can cause violations in their expected properties. Because of this, it is useful to have reliable automated testing frameworks that can simulate Byzantine behaviour to make bug detection easier. In this paper, we examine the performance of the ByzzFuzz BFT testing tool using our implementation of the HotStuff protocol. We describe the design choices necessary to create a working HotStuff implementation. Then we purposefully introduce implementation flaws to evaluate the behaviour of ByzzFuzz with different parameters and mutation scopes. We compare its performance to that of a baseline random fault injection scheduler. Our results show that it was able to detect the introduced bugs using either process or network faults. ByzzFuzz's partition-based network faults were more effective at detecting bugs than the 'Random' scheduler's network faults. For process faults, we were unable to register significant differences in performance possibly due to HotStuff's simplistic pipelined structure. In our tests, any-scope mutations performed better than their small-scope counterparts for the same configuration. This could be attributed to the nature of the selected faults and HotStuff's pipelined structure.

1 Introduction

Byzantine Fault[1] Tolerant (BFT) protocols allow the non-faulty participants in a distributed system with n nodes to reach consensus even if f of the participating nodes are Byzantine (malicious or unreliable). This is important for situations where either not all of the participants are trusted or high-availability is important. Example applications include keeping track of a distributed ledger for a cryptocurrency, and ensuring aircraft flight controls remain operational even if some system parts are malfunctioning.

Over the years there have been many proposals for practical BFT protocols, the first of which, PBFT[2], was developed in 1999. Since then there have been many other proposals for improved practical BFT protocols. Examples are Zyzzyva[3], BFT-SMaRt[4], Tendermint[5], hBFT[6]. However, due to communication complexity, most of these protocols are not easily scalable. A more recent protocol which attempts to solve this problem is the HotStuff[7] protocol, which was developed in 2019. It is the first BFT protocol that enables a correct leader to drive the protocol to consensus at the pace of actual network delay and with communication complexity that is linear in the number of participants in the system. HotStuff is also used as the basis of practical production systems such as Facebook's LibraBFT/DiemBFT[8].

Although BFT protocols aim to be resilient to Byzantine faults, errors in the protocol design, implementation bugs or network faults can cause violations of the protocol's expected properties. Many of the protocols mentioned above have been shown to have design faults[9] [10]. Despite the correctness proofs it is not uncommon to find previously undiscovered faults even years after a protocol has been deployed and is widely used in practice. This is why rigorous systematic testing is needed to increase the reliability of fault detection. A practical way to achieve this is through randomized testing tools such as ByzzFuzz [11] - a recently designed randomized testing tool for finding design or implementation faults in BFT algorithms. It simulates process faults using round-based structure-aware mutations that take into account the expected structure of the message to avoid parsing errors. It can also simulate network faults by partitioning the replicas into non-overlapping sets that cannot communicate with each other. ByzzFuzz has already successfully identified both already known as well as new unknown faults in the design or implementation of other protocols such as PBFT and Ripple.

To evaluate the performance of such testing methods we have implemented HotStuff in Java and have integrated it with the ByzzBench BFT protocol benchmarking suite. We intentionally introduce various flaws in our implementation to evaluate the performance of ByzzFuzz and a baseline randomized testing strategy that does not exhibit ByzzFuzz's properties. We also compare the performance of mutations with a different scope.

In this paper, we go over the relevant background information in Section 2, describe our methodology in Section 3, the implementation details and decisions made during the implementation process in Section 4, the experimental setup in Section 5 and summarize our results in Section 6. Then using the obtained empirical data we answer the following research questions:

- **RQ1** - Can ByzzFuzz find any bugs in our implementation of the HotStuff 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 HotStuff protocol?

Finally, we discuss what our findings mean for BFT protocol implementation and testing and we present our conclusions in Sections 7 and 8.

2 Background

In this section we describe some of the related work, how ByzzFuzz works, the relevant details of the HotStuff protocol and its variants as well as the properties which will be used to evaluate the correctness of our simulated executions.

2.1 Related work

ByzzFuzz has already been tested with implementations of other protocols. It has successfully found both liveness and safety violations on a PBFT implementation, a potential liveness violation in a Tendermint implementation, a previously known termination violation and a new unknown agreement violation in Ripple.

Twins [12] is another BFT testing method which simulates process faults by duplicating a correct node and creating a *twin*. Together they both act as a single node in the network even though they may have different internal states. Process faults are simulated by changing which twin sends/receives messages. It uses similar to our approach to evaluate the testing framework's performance where bugs are intentionally introduced in an otherwise assumed to be correct implementation.

Other BFT testing algorithms include: LOKI [13] - a fuzzing framework which has found faults in several BFT protocols including DiemBFT, BFTDiagnosis [14] - a framework which can inject predefined malicious behaviours in BFT protocols, Fluffy [15] - tool for finding consensus bugs in Ethereum using fuzzing.

A recent paper by Decouchant et al. [16] investigates the performance of 3 different liveness checking methods (temperature, lasso and timeouts) on 3 protocols from the HotStuff family (Basic HotStuff, Sync HotStuff and 2-Phase HotStuff). We do not use any of these methods for our evaluation. Instead, in Section 4.2 we introduce our own liveness invariant which is specifically tailored to our Event-Driven HotStuff implementation.

2.2 ByzzFuzz

ByzzFuzz is a randomized testing algorithm for BFT protocols. It relies on several heuristics to generate process and network faults. The main features of ByzzFuzz are that the faults it generates are *fault-bounded* - there are a limited number of process or network faults applied during each run, *round-based* - Faults are applied to a specific round instead of randomly mutating or dropping messages. The process faults are *structure-aware* - they take into account the message structure when applying mutations and *small-scope* - the mutations are based on small changes of the original values rather than replacing them with arbitrary values. ByzzFuzz also applies network faults. A network fault is applied for a specific round during which the replicas are distributed among random non-overlapping network partitions such that messages between replicas in different partitions are always dropped.

ByzzFuzz uses only 3 parameters for its scenario configuration - `numRoundsWithProcessFaults` - the bound on the rounds with process faults, `numRoundsWithNetworkFaults` - the bound on the rounds with network faults and `numRoundsWithFaults` - the bound on the round faults (no faults are inserted after this round).

2.3 The HotStuff protocol

HotStuff is a leader-based partially synchronous[17] BFT SMR protocol. The core properties of HotStuff which together differentiate it from previous practical BFT protocols are:

- **Linear View Change** - During each view, a correct leader, even during a view change, needs to send only $O(n)$ messages to make progress.
- **Optimistic Responsiveness** - A correct leader needs to wait only for $n - f$ messages to guarantee that it can create a proposal which will make progress.

These two properties are achieved through the use of Quorum Certificates (QC) which require only $n - f$ valid signatures and by introducing a third phase which removes the need for waiting for maximum network delay (compared to protocols like Tendermint[5] and Casper[18] that necessitate only 2 phases before a decision can be made but require waiting for maximum network delay).

Each replica maintains a blockchain of nodes where each node contains a client command, a QC, a reference to its parent and a height corresponding to the view number during which it was proposed. The protocol makes progress in *views*, where each view has a unique monotonically increasing number and a replica designated as leader.

The protocol works under the assumption of a point-to-point, authenticated and reliable network. It adopts the partial synchrony model of Dwork et al. [17], where there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all transmissions between two correct replicas arrive within time Δ.

Basic HotStuff

The 'Basic' version of the protocol uses 4 message types and operates in 4 core phases:

- **PREPARE** - The leader collects $n-f$ `NEW-VIEW` messages to determine the highest QC. Creates a new node and sends it to all n replicas in a `PREPARE` message. Each replica votes for the proposal in a `PREPARE-VOTE` message to the leader if it determines that the proposal is safe to accept.
- **PRE-COMMIT** - The leader collects $n - f$ `PREPARE-VOTE`s and uses them to create a QC_{PREPARE} which is sent to all n replicas in a `PRE-COMMIT` message. Each replica votes for it in a `PRE-COMMIT-VOTE` message.
- **COMMIT** - The leader collects $n - f$ `PRE-COMMIT-VOTE`s and uses them to create a $QC_{\text{PRE-COMMIT}}$ which is sent to all n replicas in a `COMMIT` message. Each replica votes for it in a `COMMIT-VOTE` message and becomes locked on it.
- **DECIDE** - The leader collects $n - f$ `COMMIT-VOTE`s and uses them to create a QC_{COMMIT} which is sent to all n replicas in a `DECIDE` message. Each replica executes the client's command associated with the proposed node and enters the next view.

Chained HotStuff

Because all 4 phases of Basic HotStuff have the same structure they can be combined into a single GENERIC phase for each view. This way the process of making a decision can be pipelined. With this approach, it takes 4 views to make a decision. The GENERIC phase in each view simultaneously serves as each of the 4 Basic HotStuff phases for the 4 chained nodes. During each view, the leader makes only one proposal and each replica casts only one vote. Appendix A contains a visualisation.

Event-Driven HotStuff

Event-Driven HotStuff further generalizes Chained HotStuff by separating the liveness mechanism from the safety logic. It also relaxes the direct ancestry constraints for performing the PRE-COMMIT and COMMIT procedures on chained nodes.

2.4 Correctness properties

To evaluate the correctness of each experimental scenario we consider the following safety and liveness properties:

- **Agreement** (safety) - Correct replicas do not make conflicting decisions.
- **Termination** (liveness) - All correct replicas eventually reach a decision.

3 Methodology

Our research is primarily experimentation-based. To answer the research questions described in Section 1 we conduct experiments using our implementation of the HotStuff protocol with ByzzFuzz and a baseline 'Random' fault scheduler.

A precondition for our experiments is to create an implementation of the HotStuff protocol. For the implementation, we have chosen to use the ByzzBench BFT protocol benchmarking suite to make running tests and investigating faults easier. ByzzBench can simulate multiple replicas concurrently. It contains a web-based GUI which allows us to monitor the execution and internal state of a simulation in real-time. It also allows us to easily create experimental configurations and to run a large number of experiments using ByzzFuzz. ByzzBench's replica simulation logic is written in Java. For the purpose of compatibility, our HotStuff implementation also needs to be written in Java.

There are multiple different versions of the HotStuff protocol described in its original paper. They all share the same core properties. For this experiment, we have chosen to implement the 'Event-Driven HotStuff' variant described under the 'Implementation' section of the HotStuff paper, because it is what the authors recommend for practical implementations. However, the high-level algorithm description does not clearly specify all the necessary details needed to create a working implementation of HotStuff and leaves a lot of room for the reader to make decisions which impact the protocol behaviour. The specific choices we have made are described in the next section.

We create a working baseline implementation that contains no detectable flaws and several intentionally buggy variations of it. The baseline implementation is created by closely following the description of the safety mechanism in the HotStuff paper and when necessary by making decisions which aim to prevent liveness violations. We verify that it does not contain faults detectable by our invariants. Then we create variations of our it by purposefully introducing bugs. This would give us a better understanding of how the two schedulers and the two mutation types behave when faced with different violations.

Finally, we use our implementations to run numerous simulated scenarios. We create different experimental configurations for both schedulers with various parameters which determine the amount of dropped messages, process and network faults and the scope of the mutations. For each simulated scenario, we use our correctness invariants to detect violations. We use the empirical data collected this way to determine the usefulness of ByzzFuzz in isolation, then we compare its performance to the 'Random' scheduler and evaluate the performance of small-scope mutations compared to any-scope mutations.

4 Implementation

In this section we describe the decisions necessary to make while implementing the protocol, the message structures and how we evaluate the protocol's correctness properties.

4.1 Implementation decisions

The HotStuff paper and the pseudocode provided in it only describe the core features of the protocol needed to guarantee linear view change and optimistic responsiveness which differentiate it from other practical BFT protocols. It acts as a more of guideline on how to create an SMR protocol exhibiting these properties rather than a description of a complete protocol. Because of this decisions that affect the protocol's properties need to be made and additional functionality needs to be implemented for the protocol to work in practice. This includes:

Replica catch-up mechanism - A catch-up mechanism is necessary to guarantee liveness in the event of a replica falling behind. If a correct replica is missing a node it would not be able to process any nodes descendent from it and as a result, it would be unable to participate in any subsequent view. This would prevent the other correct replicas from making progress because they would lack quorum. In the original paper, the necessity of such a mechanism is mentioned, but a description of how it would be implemented is omitted. To solve this problem we have implemented a catch-up mechanism which introduces 2 additional message types - one for requesting a node and one for providing a node.

The catch-up mechanism we have implemented works as follows: Whenever a replica receives a message which references the digest of a node missing from the replica's node list, the message is moved to a buffer instead of being processed. Then the replica sends an `ASK` message containing the digest of the missing node to the replica which sent the message. If the replica which receives the `ASK` message has a node which matches the digest it sends a `TELL` message containing the missing node. Upon receiving a `TELL` message the replica adds the node to its set of known nodes and processes all buffered messages which are dependent on it.

This guarantees that a replica cannot enter an invalid internal state because of a missing node since no message will be processed until the replica has all of the necessary ancestor nodes. This includes `TELL` messages as well so even if multiple ancestor nodes are missing the replica will remain in a valid state by sending and receiving a chain of `ASK` and `TELL` messages dependent on each other. If the `TELL` message is lost the messages dependent on it will remain buffered indefinitely. This will not cause a liveness violation because as long as all other system properties hold the replica will eventually receive a message referencing either the same or higher node. Even if the system is unable to make progress in the current view a new view timer will eventually cause a correct replica to send a `NEW-VIEW` message to the replica with missing node(s). This will cause the replica to send another `TELL` message which will give it another opportunity to catch up.

Client requests de-duplication - A client sends a command to all replicas. Each replica stores it in a list of pending client commands. During the replica's turn to propose a node, it selects a command which is not already included in any of the node's ancestors. Once a node containing the command becomes committed it is removed from the list of pending commands. For our simulations, we have only one client so to avoid liveness issues due to lack of requests we allow replicas to propose a node containing a client request referenced by its ancestors, but is not committed. We do not register this as a violation because this mechanism is not subject to our evaluation.

Leader election - The logic necessary to determine the leader is entirely omitted from the description of HotStuff, however, it is crucial for guaranteeing liveness. While the quorum mechanism guarantees safety a poorly chosen leader rotation schedule could prevent progress from being made indefinitely. Because of the pipelined structure of Event-Driven HotStuff and the two-chain direct ancestry requirement the protocol requires 4 consecutive views to be successful for progress to be made. This is affected by the handling of failed rounds during which a QC could not be obtained. Because the chain is extended with dummy nodes to ensure that the height is equal to the current view number anytime a leader fails to obtain a QC the chain is interrupted. If the chain could be extended directly in non-successive views then any leader rotation schedule would suffice. However, because of the $viewNumber = proposalHeight$ requirement if the leader schedule does not guarantee the existence of 4 consecutive views led by non-faulty leaders a single faulty leader would be able to prevent progress from ever being made. This prevents us from using a simple round-robin schedule where the leader is changed every view because in this case liveness could not be guaranteed with only $2f + 1$ correct replicas. What this example highlights is that the leader schedule affects the number of correct replicas required to guarantee liveness. Because of this, we choose to use a round-robin schedule where each replica remains leader for 4 consecutive views instead of only one. This ensures that as long as there is at least one correct replica there will always be a chain of at least 4 correct consecutive leaders.

Event-Driven HotStuff Pacemaker - The HotStuff paper provides a code skeleton for the *pacemaker* liveness mechanism of Event-Driven HotStuff, but many details such as when a leader should propose a new node, when replicas enter the next view and what happens after a new-view timeout are missing. To keep it consistent with the other variants we have decided to propose a node whenever the leader either collects a quorum of votes and forms a QC for the previously proposed node or whenever it collects a quorum of NEW-VIEW messages indicating a failure of the previous leader.

Message validation - We also validate the content messages when they are received. This includes checking the validity of the quorum certificates, making sure the sender of a proposal is a leader, the height of the proposed node matches the message view, and that the proposal's view number matches that of the current view.

Handling failed rounds - When a replica fails to obtain a QC it will be unable to make progress. Eventually, a `NEW-VIEW` timeout will be triggered, and each replica will send a `NEW-VIEW` message to the next leader and will move on to the next view. The next leader collects a quorum of `NEW-VIEW` messages instead of votes and proposes a node which references its highest known QC. The node referenced in the QC is extended with (implicit) dummy nodes up to the height of the current view. A dummy node is a node which does not reference a QC and does not carry a client request. It serves only as an indicator that the proposer does not have a QC for that height. For the sake of simplicity instead of creating actual dummy nodes we simply link the proposal directly to the node with the highest QC. This means that nodes with non-consequent heights are linked as if they are in a parent-child relation. However when verifying direct ancestry in the `update` procedure we add the requirement that the height of a direct ancestor must be exactly one plus the height of its parent. Otherwise, we assume the existence of implicit dummy nodes.

4.2 Correctness invariants

We use the following correctness invariants for the purpose of detecting whether one of the protocol's properties has been violated.

Agreement - We keep track of each replica's commit log and compare them. If they differ we terminate the simulation due to agreement violation.

Termination - Because each replica has a `NEW-VIEW` timer. Messages will continue to be exchanged even if no actual progress is made. This means that we cannot simply check whether there are no more scheduled events which makes detecting liveness violations more difficult.

However, as described in theorem 4 of the HotStuff paper [7] the protocol should guarantee that "After GST, there exists a bounded time period T_f such that if all correct replicas remain in view v during T_f and the leader for view v is correct, then a decision is reached." In practice, this means that if all valid replicas remain in the same view for a sufficiently long time for all messages sent up to this point to be delivered and the leader is not faulty progress will be made. We can verify whether these conditions hold and check whether progress has been made to determine whether the termination property has been violated. The assumption of GST having passed serves to ensure that the protocol will behave synchronously. So instead of waiting for GST we simply observe whether the replicas have remained synchronized for a sufficiently long period.

Because the theorem mentioned above refers to the Basic HotStuff variant which performs all 4 protocol phases in a single view we need to consider 4 consecutive views for the pipelined Event-Driven HotStuff. For each view, we check whether all replicas have been simultaneously in the same view. We also check whether there exists a `NEW-VIEW` timeout which was triggered during the view before all messages queued in the network were delivered. If so then we do not consider the system to be synchronized during this view as this would correspond to either asynchronous network behaviour before GST, which would violate our assumption of synchronization, or an insufficient timeout smaller than the actual network delay after GST, which would violate the as-

sumption that correct replicas remain in v during T_f. It is possible to perform this check because unlike in a production setting, we have full control over the simulated network and we know at all times if there are undelivered messages.

We also check whether during the view there have been any network or more than f process faults applied to the replicas. If so we also do not consider the view to be valid for the termination check since it would either break the assumption of at least $n - f$ valid replicas or that of a point-to-point reliable network. When scheduling process faults we make sure that they are not applied to more than f replicas so this assumption would be violated only for scenarios where we have network faults. Although permanently dropping a message during any view should still invalidate the protocol's assumptions we still record termination violations after a view with a dropped message has passed. The protocol could still be able to recover due to the constant exchange of `NEW-VIEW` messages.

Finally, if these conditions hold after the end of the last view we check whether any new nodes have been committed. If not we assume that there has been a liveness violation. However, this does not guarantee that all liveness violations will be caught. Because both ByzzFuzz and the 'Random' scheduler trigger timeouts probabilistically (the size of the timeout has no effect on whether a message will be delivered before the timeout or not) it is not guaranteed that during a particular run, replicas will remain synchronized for a sufficiently long sequence of views for all preconditions to hold.

4.3 Message types and data structures

Now we provide a summary of the serializable data structures we have implemented to facilitate the message exchange.
The two most important data structures we use are:

- **Quorum Certificate(QC)** $\langle n,\ s\rangle$ - Contains the *digest* for which the votes were collected and a *quorum signature* consisting of a list of *partial signatures* each of which is produced by a unique replica and references the same node digest as the QC.
- **Node** $\langle h,\ p,\ \langle n,\ s\rangle,\ c\rangle$ - Contains a *height* number, a *parent digest*, a *QC* object in its entirety and a *client request*.

Our implementation uses the following message types:

- **GENERIC** $\langle v,\ \langle h,\ p,\ \langle n,\ s\rangle,\ c\rangle\rangle$ - contains a *view number* and a *node* in its entirety.
- **GENERIC-VOTE** $\langle v,\ \langle h,\ p,\ \langle n,\ s\rangle,\ c\rangle,\ s\rangle$ - contains a *view number* and a *node* in its entirety and a *partial signature*
- **NEW-VIEW** $\langle v,\ \langle n,\ s\rangle\rangle$ - contains a *view number* and a *QC* in its entirety
- **ASK** $\langle n\rangle$ - contains only a *node digest*
- **TELL** $\langle\langle h,\ p,\ \langle n,\ s\rangle,\ c\rangle\rangle$ - contains only a single *node* in its entirety

5 Experimental Setup

In this section we describe how scheduler decisions are made, how process and network faults are simulated and the variations of our implementation created by intentionally introducing flaws.

5.1 Fault Scheduling

In our experiments we evaluate the performance of two fault injection strategies - ByzzFuzz and randomly injecting faults. ByzzFuzz injects faults based on its 3 parameters as described in Section 2.2. The 'Random' scheduler arbitrarily injects mutations or drops messages based on some probability.

Both schedulers behave asynchronously. Messages have the same probability of being delivered regardless of the order they were sent in. Timeouts are also triggered arbitrarily, with the condition that if the scheduler decides that a timeout will be triggered then it must select the timeout with the lowest target execution time.

5.2 Process faults

To simulate process faults we apply the following structure-aware small-scope and any-scope mutations. Table B summarizes the message properties to which we apply mutations for each message type.

Small-scope mutation
For GENERIC and GENERIC-VOTE messages we apply mutations that:
- Increment or decrement the view number and the node height - They need to both be changed simultaneously for the proposal to be considered valid.
- Substitute the parent node digest with the grandparent node digest
- Substitute the QC with the previous QC
- Substitute both the parent node and the QC
- Substitute the client request with that of the parent node

For NEW-VIEW messages we apply mutations that:
- Substitute the QC with the previous QC
- Increment or decrement the view

Any-scope mutations
For GENERIC and GENERIC-VOTE messages we apply mutations that:
- Replace the view number and the height with a random number
- Substitute the parent node digest with that of a random node
- Substitute the QC with another random QC
- Substitute both the parent node and the QC
- Substitute the client request with a random request

For NEW-VIEW messages we apply mutations that:
- Substitute the QC with another random QC
- Replace the view number and the height with a random number

5.3 Network faults

Both strategies can simulate network faults. The ByzzFuzz scheduler introduces network partition faults according to the `numRoundsWithNetworkFaults` parameter. The 'Random' scheduler arbitrarily drops messages based on a specified probability. It is important to note that HotStuff works under the assumption of a point-to-point reliable network so any amount of permanently dropped messages breaks this assumption.

5.4 Introducing protocol flaws

For our evaluation we use our initial implementation as a baseline, assumed to be correct at least according to our invariants. We then create several buggy variations of it in which we have purposefully introduced flaws.

- **Lowering the quorum** - We change the minimum number of valid replicas from $n-f$ to f. Only f votes are enough to create a QC and only f NEW-VIEW messages are enough to move on to the next proposal.
- **Not verifying whether the proposal height matches the current view number** - This is one of the message validation checks, referenced in Section 4.1, we have introduced. Whenever a replica receives a GENERIC message it checks whether it is intended for the current view. If not the message is discarded. We remove this check and perform a separate evaluation without it.
- **Non-monotonically increasing $\mathbf{b_{exec}}$** - In the 'Implementation' section of the HotStuff paper the b_{exec} variable is introduced. It is meant to keep track of the last executed node. The `commit` procedure is performed only if the node to be committed has greater height than the previous committed node b_{exec}. However in the provided pseudocode it is always updated in the `DECIDE` procedure regardless of whether the `commit` procedure was successful. This could lead to b_{exec} going back to a previous node which could cause the same node to be committed more than once. To avoid this in our baseline implementation we perform the same height check as in the `commit` procedure before updating b_{exec}. We also perform a separate evaluation without this check.

6 Results

In this section we present the results of our experiments first with the baseline non-faulty implementation and later with the flawed versions. Then we analyse the results and give answers to the research questions.

6.1 Parameters

The ByzzFuzz scheduler runs simulations with number of process faults and number of network faults as parameters. They are represented by the p and n columns respectively, while the r column corresponds to the maximum round bound.

For our experiments with the 'Random' scheduler we use the following parameters: m - maximum number of mutations to inject, d - maximum number of messages to drop, mW - Weighted probability the scheduler mutates a message, dW - Weighted probability the scheduler drops a message.

We simulate 1000 scenarios for each configuration. The SS and AS columns represent small-scope and any-scope mutations. A pair of cells under each column represents a separate experiment consisting of 1000 scenarios.

6.2 Baseline implementation

No liveness or agreement violations were discovered. For brevity we omit explicitly listing all baseline configurations. For any configuration listed in the 'Flawed implementations' section we have performed the same experiment for the baseline implementation with 0 agreement or termination violations being detected. Since no faults were discovered during these experiments there is no difference in the observed performance of ByzzFuzz, the 'Random' scheduler, small-scope and any-scope mutations. Because of this it is not possible to draw any conclusion form the baseline implementation alone.

6.3 Flawed implementations

We now examine the test results produced using the various faulty protocol versions described above.

Lower quorum

The results of evaluating the implementation where the quorum is set to f for both the 'ByzzFuzz' and 'Random' schedulers are described in Tables 1 and 2 respectively.

Table 1: 'Low quorum' implementation 'ByzzFuzz' scheduler results

			SS		AS	
p	n	r	A	T	A	T
1	0	20	0	0	4	0
2	0	20	1	0	10	0
3	0	20	1	0	16	0
4	0	20	2	0	20	0
5	0	20	2	0	24	0
10	0	20	6	0	55	0
5	5	20	12	0	24	0
10	10	20	46	0	85	0

p	n	r	A	T
0	0	0	0	0
0	1	20	3	0
0	2	20	1	0
0	3	20	2	0
0	4	20	5	0
0	5	20	13	0
0	10	20	51	0
0	1	10	1	0
0	2	10	1	0

p	n	r	A	T
0	3	10	1	0
0	4	10	8	0
0	5	10	26	0
0	10	10	127	0
0	1	5	0	0
0	2	5	0	0
0	3	5	8	0
0	4	5	18	0
0	5	5	30	0

Table 2: 'Low quorum' implementation 'Random' scheduler results

				SS		AS	
m	d	mW	dW	A	T	A	T
1	0	5	0	0	0	4	0
2	0	5	0	1	0	14	0
3	0	5	0	3	0	18	0
5	0	5	0	0	0	39	0
10	0	5	0	4	0	49	0
15	0	5	0	2	0	50	0
20	0	5	0	4	0	39	0
5	5	5	5	1	0	27	0
10	10	5	5	3	0	28	0

m	d	mW	dW	A	T
0	0	0	0	0	0
0	1	0	5	0	0
0	2	0	5	0	0
0	3	0	5	0	0
0	4	0	5	0	0
0	5	0	5	0	0
0	10	0	5	4	0
0	15	0	5	4	0
0	20	0	5	5	0
0	25	0	5	7	0

We were able to detect agreement violations using both schedulers. There is no significant difference in their performance when considering process faults. ByzzFuzz's network faults appear to perform better than the randomly dropped messages. Their performance improves significantly for scenarios where the round bound is low because they are more likely to cause network partitions in successive views. Any-scope mutations consistently outperform small-scope mutations for finding agreement violations. With the quorum being this low any network partition would cause agreement violation. A large amount of dropped or ignored messages would also prevent the replicas from keeping up with each

other and following the same non-diverging chain. Any-scope mutations are both more likely to produce invalid messages which would be discarded (effectively acting as a dropped message) and carry a QC which is more distant from the original high QC so they are more likely to prevent the replicas from catching up to the highest branch. The best-performing way to detect this fault is to partition the network. The network needs to remain partitioned for several consecutive views so scenarios where the round bound is lower perform better.

Skipping proposal height validation

All message validation, including this check, is omitted from the pseudocode in the HotStuff paper. We examine the results of our experiments without it in Tables 3 and 4.

Table 3: 'No proposal height validation' implementation 'Byzz-Fuzz' scheduler results

			SS		AS	
p	n	r	A	T	A	T
1	0	20	0	0	0	0
2	0	20	0	0	0	0
3	0	20	0	0	0	0
5	0	20	0	0	0	0
10	0	20	0	0	0	2
15	0	20	0	0	0	2

			SS		AS	
p	n	r	A	T	A	T
20	0	30	0	0	0	5
30	0	40	0	0	0	7
10	1	20	0	0	0	2
15	1	20	0	0	0	2
20	1	20	0	0	0	0
30	1	40	0	0	0	5

p	n	r	A	T
0	0	0	0	0
0	1	20	0	0
0	10	20	0	0
0	20	30	0	0

Table 4: 'No proposal height validation' implementation 'Random' scheduler results

				SS		AS	
m	d	mW	dW	A	T	A	T
1	0	5	0	0	0	0	0
5	0	5	0	0	0	0	0
10	0	5	0	0	0	0	1
15	0	5	0	0	0	0	0
20	0	5	0	0	0	0	0
25	0	5	0	0	0	0	1
30	0	5	0	0	0	0	1

m	d	mW	dW	A	T
0	0	0	0	0	0
0	1	0	5	0	0
0	2	0	5	0	0
0	3	0	5	0	0
0	4	0	5	0	0
0	5	0	5	0	0
0	10	0	5	0	0
0	20	0	5	0	0

We are unable to detect any violations with either scheduler when the process fault count is low. In both cases, as the number of process faults grows we are able to detect some termination violations. Both are detected only with any-scope mutations. The violations detected all share the same root cause. Because replicas do not check whether they are currently in the view of the incoming proposal they will process it regardless of its height. This cannot cause an agreement violation, because the safety logic described in the 'Event-Driven' HotStuff algorithm is unaffected. Replicas will continue to vote on proposals only if they extend the currently locked node and have monotonically increasing height. However, this leaves the protocol open to an attack on liveness. A malicious faulty node which knows the leader schedule can determine in which future view it will be the designated leader. With this information, it can construct a proposal which uses its currently valid `highQC` and extends the currently locked node but has a much larger height. The proposal will be valid as long as the malicious replica selects a view for which it is the leader. When a valid replica receives this proposal it will vote for it and will set its variable which tracks the height of the last voted node to the proposal's height. As a consequence, the correct replica will decide not to vote for any proposal with a lower height so it will be unable to participate until the rest of the replicas reach the view of the malicious replica's proposal. This view number could be much higher than the current view and an attacker would be able to continuously send proposals with ever-increasing heights to prevent the system from making progress indefinitely.

Any-scope mutations are better suited to detect this fault because they can set the view number of an otherwise valid GENERIC message to an arbitrary number which could be much higher than the current view. Because we are using implicit dummy nodes this would be equivalent to a proposal that extends the current locked node with many dummy nodes. As long as the message sender happens to be the leader for the randomly selected view number this would emulate the malicious behaviour described above. Network faults and small-scope mutations were unable to reproduce this fault with any scheduler.

Non-monotonically increasing $\mathbf{b_{exec}}$

The results of the evaluation performed on this variant are presented in Tables 5 and 6.

Table 5: 'Non-monotonically increasing b_{exec}' implementation 'ByzzFuzz' scheduler results

			SS		AS	
p	n	r	A	T	A	T
1	0	20	3	0	83	0
2	0	20	5	0	137	0
3	0	20	12	0	172	0
4	0	20	16	0	223	0
5	0	20	23	0	245	0
1	1	20	2	0	41	0
2	1	20	7	0	98	0
3	1	20	7	0	135	0
4	1	20	17	0	173	0
5	1	20	22	0	202	0

p	n	r	A	T
0	0	0	0	0
0	1	20	0	0
0	2	20	0	0
0	3	20	1	0
0	4	20	0	0
0	5	20	1	0
0	6	20	1	0
0	7	20	0	0
0	8	20	0	0
0	9	20	0	0
0	10	20	0	0

Table 6: 'Non-monotonically increasing b_{exec}' implementation 'Random' scheduler results

				SS		AS	
m	d	mW	dW	A	T	A	T
1	0	5	0	1	0	89	0
2	0	5	0	5	0	144	0
3	0	5	0	9	0	224	0
4	0	5	0	23	0	309	0
5	0	5	0	16	0	316	0

m	d	mW	dW	A	T
0	0	0	0	1	0
0	1	0	5	2	0
0	2	0	5	2	0
0	3	0	5	1	0
0	4	0	5	1	0
0	5	0	5	0	0

We were able to detect agreement violations with both scheduling methods using process faults. There does not seem to be a significant difference between the performance

of ByzzFuzz's round-based faults and random faults. Network faults do not appear to have any effect on the protocol. The few bugs found using only network faults can be attributed to the asynchronous scheduling where a leader happens to send an old QC because it has fallen behind. Any-scope mutations, on average, are around 15 times more likely to cause this agreement violation compared to small-scope for the same configuration.

6.4 Analysis

Given the results described above we answers the 3 research questions.

RQ1 - Can ByzzFuzz find any bugs in our implementation of the HotStuff protocol?

To answer RQ1 we look at the experimental results of the ByzzFuzz scheduler in isolation. There were no bugs found in our baseline implementation. When evaluating the other faulty implementations ByzzFuzz was able to detect the introduced bugs. For the implementation with a lower quorum, it was able to detect the agreement violations both through process and network faults with the network partitions of ByzzFuzz being particularly effective. ByzzFuzz was also able to detect the 'No proposal height validation' liveness violation and the 'Non-monotonically increasing b_{exec}' agreement violation.

RQ2 - How does the bug detection performance of ByzzFuzz compare to a baseline testing method that arbitrarily injects network and process faults?

When using only process faults there was no significant difference in the performance of the two methods. HotStuff's simplistic pipelined structure and the lack of any resend mechanism possibly negates some of the benefits of applying ByzzFuzz's round-based mutations. For the 'low quorum' implementation ByzzFuzz was able to more reliably detect the flaws due to its network partitions.

RQ3 - How do small-scope and any-scope message mutations of ByzzFuzz compare in their performance of bug detection for the HotStuff protocol?

Contrary to the assumptions of ByzzFuzz for our selected faulty implementations any-scope mutations were consistently more likely to detect the introduced bugs. Small-scope mutations performed worse for the 'low quorum' and 'Non-monotonically increasing b_{exec}' implementations and were unable to trigger the 'No proposal height validation' termination violation at all. How well small-scope and any-scope mutations perform depends on the nature of the specific implementation flaw. Although any-scope mutations perform better in our experiments further research is needed to determine which performs better for more 'natural' implementation flaws.

7 Discussion

Our work shows the importance of using a wider range of test configurations for reliable bug detection. The many instances where any-scope mutations outperformed small-scope and the lack of meaningful difference between randomly scheduled and round-based process faults show that the assumptions of ByzzFuzz that small-scope and round-based mutations perform better may not be valid for pipelined protocols such as Event-Drive HotStuff. Since the protocol requires 4 views to make progress flaws may be more easily revealed if ByzzFuzz could purposefully schedule faults in multiple successive views.

Our work also highlights some of the challenges and considerations necessary for implementing a working version of the HotStuff protocol - in particular the decisions that need to be made to avoid liveness violations. While the algorithm guarantees agreement numerous aspects of the protocol vital for liveness are entirely omitted from the original HotStuff paper. The lack of theoretical guarantees combined with the difficulty of detecting such violations leaves implementations potentially vulnerable to liveness attacks. As an example, a recent paper highlights a problem with the timer doubling view synchronization mechanism suggested by the authors of HotStuff. [19]

8 Conclusions and Future Work

In this paper, we investigated the performance of the ByzzFuzz randomized BFT protocol testing algorithm against the 'Random' baseline scheduler based on their bug-detection performance on our HotStuff implementation. We described the challenges faced when implementing HotStuff, the small-scope and any-scope mutations we apply as well as the faults we purposefully introduce in our implementation for evaluation purposes. We performed multiple experiments with different parameters for each of our implementation variants. Our results show that ByzzFuzz was able to find all injected protocol flaws. When relying on process faults its performance is consistent with the 'Random' fault scheduler, but its network partition faults may perform better in some cases. For many of our experiments, we also observed that any-scope mutations are more likely to detect the bugs we introduced than small-scope mutations.

Further experiments are necessary to provide a more complete understanding of how randomized testing strategies perform with more advanced implementations of HotStuff-like protocols. Such implementations may include a more complex leader election mechanism, client request de-duplication, and synchronization mechanism. Performing experiments with additional BFT testing tools could provide a better perspective of their comparative performance. Analysis and classification of bugs detected in production systems could advise a more adequate selection of intentionally introduced implementation flaws.

9 Responsible Research

To ensure the reproducibility of our results we have tried to follow the pseudocode described in the HotStuff paper as closely as possible. We have also documented any important implementation decisions that had to be made to achieve a working implementation. Before recording the results of

our experiments we have put effort into ensuring that the experimental setup closely matches its description in this paper. We have also closely investigated the execution logs before recording any results to ensure that the system has behaved as expected. Our implementation will be made publicly available on GitHub so anyone can try to reproduce our results.

References

[1] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: https://doi.org/10.1145/357172.357176.

[2] Miguel Castro, Barbara Liskov, et al. "Practical byzantine fault tolerance". In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186.

[3] Ramakrishna Kotla et al. "Zyzzyva: speculative byzantine fault tolerance". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, pp. 45–58. ISBN: 9781595935915. DOI: 10.1145/1294261.1294267. URL: https://doi.org/10.1145/1294261.1294267.

[4] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. "State machine replication for the masses with BFT-SMART". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 355–362.

[5] Ethan Buchman. "Tendermint: Byzantine fault tolerance in the age of blockchains". PhD thesis. University of Guelph, 2016.

[6] Sisi Duan, Sean Peisert, and Karl N. Levitt. "hBFT: Speculative Byzantine Fault Tolerance with Minimum Cost". In: *IEEE Transactions on Dependable and Secure Computing* 12.1 (2015), pp. 58–70. DOI: 10.1109/TDSC.2014.2312331.

[7] Maofan Yin et al. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. PODC '19. Toronto ON, Canada: Association for Computing Machinery, 2019, pp. 347–356. ISBN: 9781450362177. DOI: 10.1145/3293611.3331591. URL: https://doi.org/10.1145/3293611.3331591.

[8] Diem Team. "Diembft v4: State machine replication in the diem blockchain". In: *Diem (Libra, Novi a Facebook Project. 2021. url: https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf.(accessed: 18.11. 2022)(pages 35, 121)* (2021).

[9] Ittai Abraham et al. "Revisiting fast practical byzantine fault tolerance". In: *arXiv preprint arXiv:1712.01367* (2017).

[10] Nibesh Shrestha, Mohan Kumar, and SiSi Duan. "Revisiting hbft: Speculative byzantine fault tolerance with minimum cost". In: *arXiv preprint arXiv:1902.08505* (2019).

[11] Levin N. Winter et al. "Randomized Testing of Byzantine Fault Tolerant Algorithms". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023). DOI: 10.1145/3586053. URL: https://doi.org/10.1145/3586053.

[12] Shehar Bano et al. *Twins: BFT Systems Made Robust*. 2022. arXiv: 2004.10617 [cs.CR]. URL: https://arxiv.org/abs/2004.10617.

[13] Fuchen Ma et al. "LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols." In: *NDSS*. 2023.

[14] Jitao Wang et al. "BFTDiagnosis: An automated security testing framework with malicious behavior injection for BFT protocols". In: *Computer Networks* 249 (2024), p. 110404. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2024.110404. URL: https://www.sciencedirect.com/science/article/pii/S1389128624002366.

[15] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. "Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 349–365. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/yang.

[16] Jérémie Decouchant, Burcu Kulahcioglu Ozkan, and Yanzhuo Zhou. "Liveness Checking of the HotStuff Protocol Family". In: *2023 IEEE 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE. 2023, pp. 168–179.

[17] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (Apr. 1988), pp. 288–323. ISSN: 0004-5411. DOI: 10.1145/42282.42283. URL: https://doi.org/10.1145/42282.42283.

[18] Vitalik Buterin and Virgil Griffith. *Casper the Friendly Finality Gadget*. 2019. arXiv: 1710.09437 [cs.CR]. URL: https://arxiv.org/abs/1710.09437.

[19] Kaiwen Guo, Kexin Hu, and Zhenfeng Zhang. "Liveness Attacks On HotStuff: The Vulnerability Of Timer Doubling Mechanism". In: *The Computer Journal* 67.8 (Apr. 2024), pp. 2586–2600. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxae027. eprint: https://academic.oup.com/comjnl/article-pdf/67/8/2586/58796506/bxae027.pdf. URL: https://doi.org/10.1093/comjnl/bxae027.

A Pipelined HotStuff visualisation

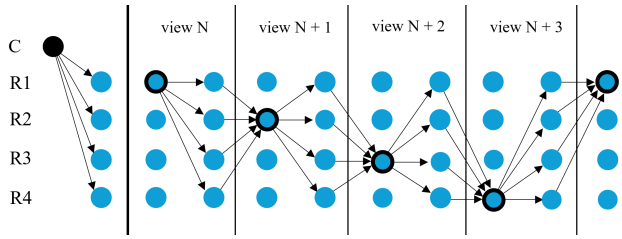

Figure 1: Chained HotStuff View Change - The leader of each view collects votes for the node proposed during the previous view

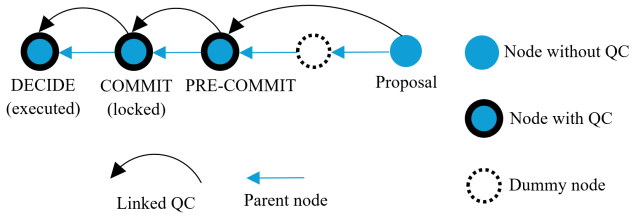

Figure 2: Node chain where each node represents different protocol phase

B Message mutation summary

Table 7: Event-Driven HotStuff message mutations

Message Type	**Mutations**
GENERIC	$\langle \mathbf{v\prime}, \langle \mathbf{h\prime}, p, \langle n, s\rangle, c\rangle\rangle$ $\langle v, \langle h, \mathbf{p\prime}, \langle n, s\rangle, c\rangle\rangle$ $\langle v, \langle h, \mathbf{p\prime}, \langle \mathbf{n\prime}, \mathbf{s\prime}\rangle, c\rangle\rangle$ $\langle v, \langle h, p, \langle \mathbf{n\prime}, \mathbf{s\prime}\rangle, c\rangle\rangle$
GENERIC-VOTE	$\langle \mathbf{v\prime}, \langle \mathbf{h\prime}, p, \langle n, s\rangle, c\rangle, s\rangle$ $\langle v, \langle h, \mathbf{p\prime}, \langle n, s\rangle, c\rangle, s\rangle$ $\langle v, \langle h, \mathbf{p\prime}, \langle \mathbf{n\prime}, \mathbf{s\prime}\rangle, c\rangle, s\rangle$ $\langle v, \langle h, p, \langle \mathbf{n\prime}, \mathbf{s\prime}\rangle, c\rangle, s\rangle$
NEW-VIEW	$\langle \mathbf{v\prime}, \langle n, s\rangle\rangle$ $\langle v, \langle \mathbf{n\prime}, \mathbf{s\prime}\rangle\rangle$