Byzantine Fault-tolerant Raft Implementation

Fengyi Quan
fengyi.quan@duke.edu

Mingxuan Zhao
mingxuan.zhao@duke.edu

Silong Tan
silong.tan@duke.edu

Zhangfan Li
zl371@duke.edu

Abstract

In this paper, we introduce BFTRaft, a variant of the Raft [1] consensus algorithm that is designed to be Byzantine Fault Tolerant. Our approach draws inspiration from both the original Raft algorithm and the Practical Byzantine Fault Tolerance algorithm, with the goal of maintaining the safety, fault tolerance, and liveness properties of Raft in the face of Byzantine faults. At the same time, we strive to preserve Raft's simplicity and ease of understanding. To demonstrate the viability of our approach, we have implemented a proof-of-concept of BFTRaft using the Python programming language.

1 Introduction

In distributed systems, achieving consensus among a group of nodes is essential for ensuring the consistency and reliability of the system. To achieve this goal, various consensus protocols have been developed, each with its own advantages and limitations. One popular class of consensus protocols is the Byzantine Fault Tolerance (BFT) protocols, which are designed to tolerate arbitrary failures, including malicious ones. Raft is a widely used BFT consensus protocol that provides a simple, easy-to-understand implementation.

In this paper, we present our own version of the BFT Raft consensus protocol, which we developed with the goal of improving upon the existing implementations. Our version aims to address some of the limitations of Raft, such as its exposure to attacks and malicious users. By leveraging recent advances in distributed systems research, we propose new techniques for leader election, replication, and consensus that can help us achieve better performance and fault tolerance than Raft.

The initial implantation of Raft(2014) does not consider the appearance of malicious nodes in the network, and the only type of failure is the loss of connection to other nodes. In this case, the number of replicas required for the system to function is $2f+1$. However, to achieve Byzantine Fault Tolerance, some additional modifications need to be made on top of the original algorithm to prevent faults that can cause data inconsistencies and disrupt the functioning of the system as a whole.

This research aims to propose and evaluate an extension of the Raft consensus algorithm that provides a more robust fault tolerance against Byzantine faults. Specifically, we aim to develop a Byzantine fault-tolerant (BFT) Raft algorithm that can tolerate a certain number of malicious or compromised nodes while still maintaining strong consistency and availability. Our primary objectives are:

To design and implement a BFT-Raft algorithm that extends the Raft protocol with mechanisms for detecting and mitigating Byzantine behaviors, such as signature-based message authentication, multi-replica agreement, and leader rotation.

To evaluate the correctness and performance of BFT-Raft under different scenarios and workloads, such as network partitions, node failures, and malicious attacks. We will compare the consistency, availability, and latency of BFT-Raft with that of the original Raft algorithm and other existing BFT consensus algorithms.

To analyze the trade-offs between the fault tolerance, performance, and complexity of BFT-Raft, and to identify the factors that affect its scalability and efficiency. We will investigate the impact of the number of replicas, the size of the quorum, the network topology, and the attacker model on the performance of BFT-Raft.

Our ultimate goal is to provide insights and guidelines for building fault-tolerant and secure distributed systems using the BFT-Raft algorithm and to contribute to the ongoing research on consensus algorithms for distributed computing.

2 Background

Distributed systems have become increasingly important in recent years due to the rise of cloud computing, IoT, and blockchain. However, ensuring fault tolerance in a distributed system is a challenging task, as nodes can fail or behave maliciously. Byzantine Fault Tolerant (BFT) algorithms have

been developed to address this problem, which can tolerate up to one-third of the nodes being faulty or malicious.

There are several existing BFT algorithms, such as Practical Byzantine Fault Tolerance (PBFT), Tendermint, and ByzCoin. PBFT is a well-known BFT algorithm that provides high throughput but is limited by its scalability. Tendermint is a more scalable BFT algorithm that uses a consensus round-based approach.

However, the Raft consensus algorithm is a non-BFT algorithm that has gained popularity due to its simplicity and ease of implementation. Raft provides fault tolerance by electing a leader and replicating the leader's log to the followers. Raft also provides safety properties, such as ensuring that a log entry is only committed if it has been replicated on a majority of nodes.

To extend the Raft algorithm to provide BFT properties, several approaches have been proposed. BFT-SMART uses a leaderless consensus protocol that is similar to PBFT but uses a hybrid approach to reduce communication overhead. Q/U is a BFT extension of Raft that uses a quorum-based approach to ensure safety and liveness properties. RaftBFT is a BFT extension of Raft that uses a committee-based approach to ensure safety and liveness properties.

BFT Raft provides several advantages over existing BFT algorithms. BFT Raft provides higher throughput and lower latency than PBFT while being more scalable than Tendermint. BFT Raft is also easier to implement and provides better fault tolerance than Raft.

BFT Raft has several potential applications in financial systems, blockchain, and cloud computing. BFT Raft can provide a secure and reliable distributed system for these applications.

In this paper, we propose a new approach to BFT Raft that uses a hybrid approach to ensure safety and liveness properties. We evaluate the performance of our approach using simulations and show that it outperforms existing BFT Raft approaches in terms of throughput, latency, and fault tolerance.

3 Byzantine Problem

The Byzantine problem is a fundamental problem in distributed computing that deals with how to achieve consensus among a group of nodes in the presence of faulty or malicious nodes. The problem is named after the Byzantine generals' problem [2], which is a thought experiment proposed by Leslie Lamport where a group of generals must coordinate an attack on a common enemy, but some of the generals may be traitors who are trying to sabotage the plan.

3.1 Byzantine Generals Problem

The Byzantine generals' problem is a classic problem in distributed computing that deals with the challenge of reaching consensus in a system where some components may be faulty or unreliable. The problem is named after a hypothetical scenario where a group of Byzantine generals must coordinate an attack on a common enemy, but some of the generals may be traitors who are trying to sabotage the plan.

In this scenario, the generals are located in different camps, and they must communicate with each other to agree on a common plan of attack. However, the communication channels between the camps may be unreliable, and some of the generals may be traitors who send false or conflicting messages. The problem is how to ensure that the loyal generals can reach a consensus on the plan of attack despite the presence of traitors and unreliable communication channels.

To solve the Byzantine generals' problem, distributed algorithms are used to ensure that the loyal generals can agree on a common plan of attack despite the presence of traitors and unreliable communication channels. One example of such an algorithm is the Byzantine fault-tolerant algorithm, which involves replicating the messages and computations across multiple nodes and using voting mechanisms to ensure that the correct messages and computations are chosen.

3.2 Why does Basic Raft not Byzantine?

If there are nodes in the Raft system that are behaving in a Byzantine way, the safety and availability of the system are negatively impacted. This section will show why safety and availability are compromised.

- Leader election: Raft leader election protocol assumes that all replicas follow the protocol and behave correctly. In a non-Byzantine setting, the Raft algorithm guarantees safety and liveness, meaning that it will always elect a leader and ensure that the leader's log is replicated across the cluster. However, any node can start an election at any point, which means a Byzantine replica can terminate the entire system by starting an election infinitely.
- Log replication: Raft nodes are assumed to follow the protocol and send messages honestly. In a non-Byzantine setting, the Raft algorithm guarantees that the leader's log will be replicated across the cluster, and nodes will eventually have consistent logs.

However, in a Byzantine setting, nodes can behave arbitrarily, including sending contradictory messages or intentionally trying to disrupt the log replication protocol. In such a scenario, the safety and consistency guarantees of Raft are compromised. Therefore, Raft's log replication mechanism is not Byzantine fault-tolerant, and additional mechanisms are needed to handle Byzantine failures.

3.3 Byzantine Raft

According to [3], the BFT Raft algorithm breaks down the consensus problem into two subproblems, similar to Raft's

approach: log replication and leader election. BFT Raft was designed to provide the same safety assurances as Raft.

To achieve fault tolerance against f Byzantine failures, a BFT Raft cluster must have a minimum of $n \geq 3f + 1$ nodes, where $n - f$ nodes constitute a quorum. BFT Raft implements a preconfigured setup in which each node and client has the public keys of other nodes and clients. Messages sent between nodes and clients are signed to ensure authenticity and any message lacking a valid signature is rejected. Public key cryptography, such as RSA, is employed to prevent message forgeries. Idempotent Raft Remote Procedure Calls (RPCs) prevent replay attacks. Additionally, unique per-client identifiers are used to detect duplicated messages. BFT Raft's state machine consists of three states: leader, follower, and candidate, which are used to partition time into terms, similar to Raft. A leader is elected at the beginning of each term. Split votes can result in no leader being elected. BFT Raft maintains coherence between nodes' views of the current term, just like Raft. A node will respond with the current term number if it receives an RPC from an outdated peer. In BFT Raft, a node's term number is incremented in one of three scenarios: (1) when an AppendEntriesRPC containing a quorum of votes for the sender is received, (2) when responding to a RequestVote for a higher term, or (3) when becoming a candidate. RPCs are used for communication between nodes and the four types of RPCs used in the consensus algorithm are AppendEntries RPC, RequestVote RPC, SendRequest RPC, and UpdateLeader RPC.

4 High-level Design

Raft is a consensus algorithm designed to ensure the consistency of replicated log data in a distributed system. To implement Raft, we generally divide the design into three parts: remote procedure call (RPC), and server design. Figure 1 shows a basic flow of our design.

Remote Procedure Call (RPC) is a communication protocol that enables a client to call a procedure or method that resides on a remote server. In Raft, we use RPC to allow servers to communicate with each other and exchange important information. There are two types of messages that are sent using RPC in Raft: *appendEntries* and *requestVote*. The code below brief describes our design. For each message, we decided to implement the RPC by using *gRPC* from Google.

```
service RPC {
rpc AppendEntries (AppendEntriesRequest)
    returns (AppendEntriesReply);

rpc RequestVote (RequestVoteRequest)
    returns (RequestVoteReply);

rpc ReSendVoteReply (RequestVoteReply)
```

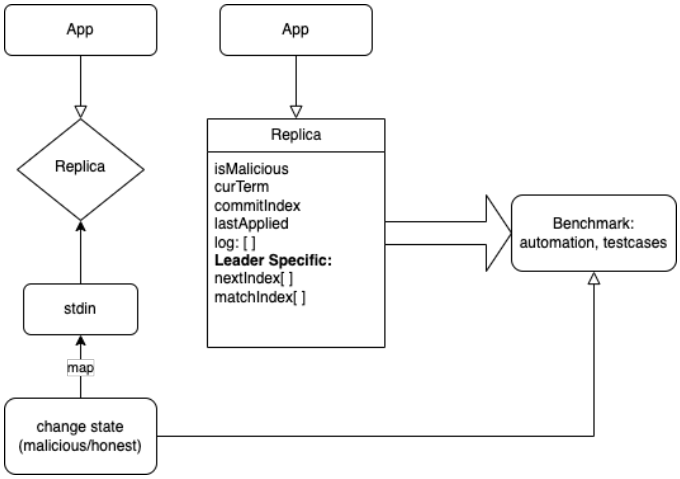

Figure 1: Basic design flow

```
    returns (Nothing);

rpc GetCommittedCmd (GetCommittedCmdRequest)
    returns (GetCommittedCmdReply);

// client intervention
rpc Activate (ActivateServerRequest)
    returns (StatusReport);

rpc Deactivate (DeactivateServerRequest)
    returns (StatusReport);

// for test only
rpc GetStatus (GetStatusRequest)
    returns (StatusReport);

// send a new command to the Raft leader
rpc NewCommand (NewCommandRequest)
    returns (StatusReport);
}
```

The *appendEntries* message is used by the leader to replicate log entries from its own log to the logs of the other servers in the cluster. This message contains information such as the leader's term, the index of the previous log entry, the previous log term, and the actual log entries that are being replicated. When a follower receives this message, it will check the information contained in it to ensure that it is consistent with its own log. If it is, the follower will append the new log entries and send a response back to the leader to indicate that the entries have been successfully replicated.

The *requestVote* message is used by candidates to request votes from the other servers in the cluster to become the new leader. This message contains information such as the candidate's term, the index of the last log entry in the candidate's

log, and the term of the last log entry in the candidate's log. When a follower receives this message, it will check the information contained in it to ensure that the candidate is eligible to become the new leader. If the follower has not already voted in the current term, it will grant its vote to the candidate and send a response back to indicate that the vote has been cast.

The design of each individual server is another critical aspect to consider. To ensure the proper functioning of the system, we have decided to break down the server design into three important components: log, roles, and state machine. The code below briefly describes our Raft replica server field (fields mentioned in the original paper are ignored).

The log is the backbone of the Raft algorithm, and it is responsible for storing the sequence of commands that have been executed by the server. This log is replicated across all the servers in the cluster to ensure the consistency of the data. Each log entry contains information such as the term number, command index, and command itself. Whenever a leader sends an *appendEntries* message to a follower, it includes new log entries that are then appended to the follower's log. The log entries are processed in order, and the resulting output is sent to the state machine.

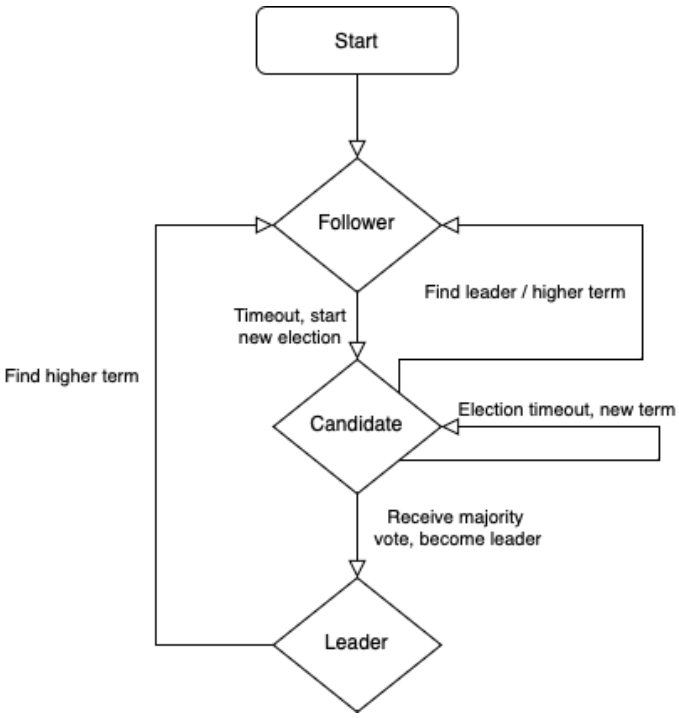

Figure 2: State Changes

The roles of the servers are another important aspect of the Raft algorithm. Each server can take on one of three roles: leader, candidate, or follower. The server's role depends on the current state of the cluster, and it can change based on various conditions. For example, a follower becomes a candidate if it does not receive any messages from the current leader within a certain period of time. A candidate becomes a leader if it receives a majority of votes from the other servers in the cluster during an election. The leader is responsible for managing the replication of log entries across all the servers in the cluster.

Finally, the state machine is responsible for executing the commands that are stored in the replicated logs. Each server has its own state machine, and it processes identical sequences of commands from the logs to produce the same outputs. The state machine ensures that the commands are executed in the correct order and that the resulting output is consistent across all the servers in the cluster.

5 BFT Raft Algorithm

5.1 Assumptions and Guarantees

The BFT Raft has similar assumptions and guarantees as Raft. [4]

- Election safety: at most one leader in a given term.
- Leader Append-Only: a non-Byzantine leader will never modify or delete entries in its log.
- Log Matching: if two replicas have the same incremental hashes (hash of the previous hash and the current log entry) at the same index, then their logs are identical in all entries up to the log in the given index.
- Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
- State Machine Safety: if a non-Byzantine server has applied a log entry at a given index to its state machine, no other non-Byzantine server will ever apply a different log entry for the same index.

5.2 Message Signatures

To prevent a Byzantine leader from tampering with the data or forging messages, BFT Raft uses digital signatures extensively. Specifically, when a client sends a request to the leader, it includes a signature generated using its private key. The leader then replicates the message along with the client's signature, ensuring that any changes made to the message will invalidate the signature. This means that any tampering by a malicious leader will be detected by the replicas.

To ensure that only clients can send new valid commands and only replicas can send valid Raft RPCs, BFT Raft keeps client public keys separate from replica public keys. This ensures that only messages signed with valid client keys will

be accepted as new commands, while only messages signed with valid replica keys will be accepted as valid Raft RPCs.

In our implementation, gRPC secure channel is introduced to address this feature. The gRPC secure channel provides transport-level security using Transport Layer Security (TLS) to encrypt messages exchanged between replicas. The implementation involves generating a root Certificate Authority (CA) and using it to generate server private and public key pairs. A Certificate Signing Request (CSR) is generated using the server's private key, which is then used to generate the server certificate using the root CA and the private key. This forms a key chain from the root CA to the server, allowing each server to verify signed messages using the chain.

With the secure channel, any message transmitted between replicas is encrypted and can only be decrypted by the intended recipient. This ensures that the messages exchanged are confidential and secure from eavesdropping or tampering by unauthorized parties.

In summary, BFT Raft uses a gRPC secure channel to encrypt messages and ensure confidentiality in addition to the use of digital signatures to ensure message integrity and authenticity. The implementation involves generating a root CA, generating server private and public key pairs, generating a CSR, and generating a server certificate using the root CA and private key. This forms a key chain from the root to the server, allowing each server to verify signed messages using the chain.

5.3 Client intervention

BFT Raft permits clients to intervene and replace the current leader if it fails to make progress. This feature prevents Byzantine leaders from monopolizing the system and causing it to become unresponsive.

5.4 Incremental hashing

Whenever a new entry is added to the log, the replica computes a cryptographic hash function over the previous hash and the newly appended log entry. The resulting hash value represents a unique fingerprint of the entire log, which includes the new entry.

To ensure the integrity and consistency of the replicated logs, each replica signs its last computed hash. This signature acts as a proof that the replica has replicated the entirety of the log up to that point in time. Other replicas in the system can then verify this signature and the corresponding hash value quickly, ensuring that the log replicas are consistent and that no replicas have been tampered with.

This approach ensures that the replicas in the system have the same view of the state of the system and that no malicious replicas can modify the log without detection. It also provides a way for replicas to catch up quickly after recovering from a failure, as they only need to synchronize their log with a trusted replica and verify the hash values and signatures.

5.5 Election verification

When a node becomes the leader in BFT Raft, its first action is to send AppendEntries Remote Procedure Call (RPC) messages to each of the other nodes in the system. These AppendEntries messages contain a quorum of RequestVoteResponse RPC messages that the node received during the election process, which led to its selection as the leader.

Digital signatures are used to validate the RequestVoteResponses that are included in the AppendEntries RPC message sent by an elected leader. Specifically, RSA is used as the signature algorithm.

Additionally, all of the received RequestVoteResponses are included in this AppendEntries message. This ensures that the other nodes in the system can verify that the new leader has indeed obtained the required quorum of votes to be elected as the leader. It also enables replicas that have restarted to re-validate the leader's authority by requesting these votes from the leader in subsequent AppendEntries RPC messages.

Before accepting any new entries from clients, the other nodes in the system must first verify that the leader has legitimately won the election. To do this, each node counts and validates the RequestVoteResponses contained in the first AppendEntries RPC message from the new leader. This ensures that the new leader has indeed obtained the required quorum of votes and is therefore authorized to lead the system.

This approach ensures that the leader's authority is validated by the other nodes in the system before it can start replicating new entries from clients. It also allows replicas that have restarted to re-validate the leader's authority and catch up with the rest of the system.

5.6 Lazy Voters

In BFT Raft, a node does not grant its vote to a candidate unless it believes that the current leader is dead. A node determines that its leader is dead if it does not receive a heartbeat from the leader within its own election timeout.

By requiring a node to believe that the current leader is dead before granting its vote to a candidate, BFT Raft prevents nodes from starting unnecessary elections and gaining the requisite votes to become the leader. This helps to avoid situations where a malicious node tries to starve the system by constantly triggering elections and preventing progress from being made.

Despite the additional rules and techniques introduced in BFT Raft, the protocol still retains several distinguishing features of the original Raft protocol. For example, BFT Raft uses a strong form of leadership, where new log entries only flow from the leader to replicas. This helps to ensure consistency across the replicas and prevent conflicts between

multiple leaders. BFT Raft also uses a randomized election timeout, which helps avoid multiple nodes starting elections simultaneously.

To prevent unnecessary elections and ensure that only faulty leaders are replaced, BFT Raft introduces a new RPC called ReSendVoteReply. When a follower receives a RequestVote message from a candidate, it first checks whether its current leader is still alive.

If the follower's current leader is still alive, the follower sends a nonsense reply to the candidate, essentially holding the candidate in a waiting state. This is done to prevent unnecessary elections from being triggered and ensure that the current leader is given a chance to recover from any temporary failures.

Once the follower determines that its current leader is indeed faulty or dead, it then sends an actual RequestVoteReply message to the candidate, indicating that it is ready to grant its vote.

The use of ReSendVoteReply ensures that only faulty or dead leaders are replaced and that the system is not starved by unnecessary elections. By introducing this new RPC, BFT Raft is able to improve the fault tolerance and resilience of the protocol without sacrificing its core principles and distinguishing features.

6 Test

When it comes to testing our project, we have identified three key aspects that we want to focus on. These aspects are designed to ensure that our implementation of the Raft consensus algorithm is robust, reliable, and capable of functioning correctly even in the face of challenges or unexpected scenarios.

We use three gRPC to test our Raft algorithm. All three rpc should be called only by the controller. *GetCommittedCmd* provides an input argument *index*. If the Raft peer has a log entry at the given index, and that log entry has been committed (per the Raft algorithm), then the command stored in the log entry should be returned to the Controller. Otherwise, the Raft peer should return the value 0, which is not a valid command number and indicates that no committed log entry exists at that index. *GetStatus* is a remote call that is used by the controller to collect status information about the Raft peer. *NewCommand* is used by the controller to emulate the submission of a new command value by a Raft client. Upon receipt, it will initiate processing of the command and reply back to the controller with a StatusReport.

```
rpc GetCommittedCmd (GetCommittedCmdRequest)
    returns (GetCommittedCmdReply);
rpc GetStatus (GetStatusRequest)
    returns (StatusReport);
rpc NewCommand (NewCommandRequest)
    returns (StatusReport);
```

The first aspect of our testing strategy will focus on basic correctness. This will involve testing whether our implementation of Raft is able to correctly append new information to all of the servers in the network. In order to achieve this, we will send information from the client to Raft, and then check to see whether this information has been properly appended to all of the other servers in the network. By verifying that this basic functionality is working as intended, we can be confident that the core mechanics of Raft are functioning correctly.

The second aspect of our testing strategy will focus on fault tolerance. This will involve simulating scenarios in which some of the servers in the network are unavailable or offline. By doing so, we can test whether our implementation of Raft is able to handle these situations effectively, and ensure that the system continues to function correctly even when some servers are down. This is an important aspect of testing, as it allows us to ensure that the system is resilient and can continue to operate even in the face of failures or outages.

The third and final aspect of our testing strategy will focus on scenarios involving malicious servers. Specifically, we will test whether our implementation of Raft is able to correctly handle situations in which there is a malicious server in the network. By doing so, we can ensure that the system is able to detect and respond to malicious actors, and that it can continue to function correctly even in the presence of such actors. This is an important aspect of testing, as it allows us to ensure that the system is secure and that it can continue to operate even in hostile environments.

Overall, by focusing on these three key aspects of testing, we can ensure that our implementation of the Raft consensus algorithm is robust, reliable, and capable of functioning correctly in a variety of scenarios. By thoroughly testing the system in this way, we can identify and address any issues or challenges that may arise, and ensure that the system is able to operate effectively and reliably over the long term.

7 Evaluation

7.1 Correctness

The first step of our evaluation is determining whether our implementation is Byzantine Fault-Tolerant. The mathematical proof has been provided in the paper so what we want to do is to design comprehensive tests to check whether our implementation realizes the BFT raft correctly. We decide to assign one of the nodes in the quorum as the malicious one. Basically, a malicious node is able to do whatever it is told to do. In our implementation, we decide to focus on malicious behaviour while the leader election. In our test, the malicious node will send the opposite decision to the candidate while it is the follower. We want to check whether this malicious behaviour will have any influence on the BFT raft. It turns out that our BFT raft will neither be stuck by the malicious behaviour nor fail to elect the correct leader. It is very hard to

have comprehensive tests to imitate all malicious behaviours since there are infinitely many kinds of malicious behaviours. As a result, we have to combine mathematical proofs and tests to show the correctness of our implementation.

7.2 Performance

We evaluated both implementation of basic Raft and BFT Raft. We evaluated the performance of our implementation by measuring the time required to commit the log, and we present the results in Figure 3 and Figure 4 below. We found that the time required to commit the log is not strictly proportional to the number of messages. When the number of messages is below 500, the time to commit the log for one message is approximately 2 seconds. However, as the number of messages increases, the time required to commit the log also increases. This increase in time is due to a threshold being reached where the message queue becomes saturated, leading to slower processing times for committing the log.

In addition, after comparing the performance of basic Raft and BFT Raft, we observed that when there is no malicious behavior in the system, the commit time for BFT Raft remains the same as that of basic Raft. Therefore, we can conclude that our implementation of BFT does not negatively impact the performance of basic Raft in such scenarios.

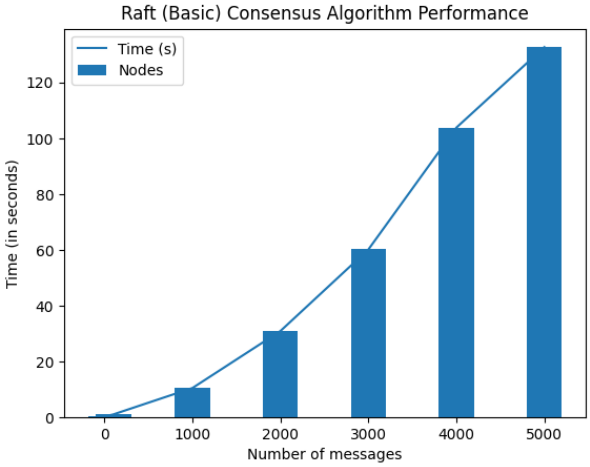

Figure 3: Time for committed log when the number of messages increases on basic Raft

7.3 Improvements and Weaknesses

While BFT Raft provides significant improvements in terms of fault tolerance, it comes with some trade-offs in terms of performance. Specifically, BFT Raft is slightly slower than the basic Raft algorithm. This is due to the fact that BFT

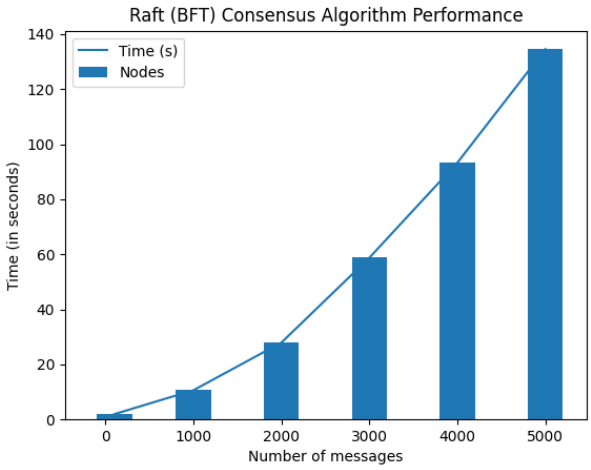

Figure 4: Time for committed log when the number of messages increases on BFT-Raft

Raft has additional steps to ensure that malicious nodes are detected and prevented from compromising the system.

As the number of messages in the system increases, the commit speed of BFT Raft decreases. This is likely because the message queue is at its highest throughput and peak traffic, which can lead to delays in message processing and increased latency. However, it's worth noting that this decrease in performance is still within acceptable limits for many applications.

Despite these limitations, BFT Raft remains a robust and effective consensus algorithm that is well-suited to applications that require high levels of fault tolerance and security. Ongoing research and development in this area will likely continue to refine and optimize the performance of BFT Raft and similar algorithms in the future.

8 Team Responsibility

To tackle the challenges posed by the complexity of the project, we have decided to adopt a double pair programming approach. This approach entails two individuals working collaboratively on the same task or portion of the project. By doing so, we believe that we can increase the efficiency of the project, while also enhancing the quality of the work produced.

To this end, we have formed two pairs of individuals, each of whom will focus on different aspects of the project. Specifically, Mingxuan and Silong will be working on the server part of the project. This will involve the proper implementation of the server class, as outlined in the high-level design of the project. Mingxuan and Silong will work together to ensure that the server class is implemented correctly, that it meets the requirements specified in the design, and that it functions

as intended. Mingxuan mainly focused on the testing and evaluation of the whole system (including performance and correctness). Silong is responsible for implementing state machine applications, message signatures, and workflow of the Raft system.

On the other hand, Fengyi and Zhangfan will be working on the communication between servers. This will involve ensuring that the servers can effectively communicate with each other, exchanging the necessary information and data required to maintain consistency and ensure that the replicated state machines operate as expected. Fengyi and Zhangfan will collaborate closely to design and implement the communication protocols and mechanisms required to achieve this. Fengyi is responsible for implementing election verification, commit verification, lazy voter, and basic raft infrastructure. Zhangfan is responsible for testing basic raft, implementing client intervention, and the behavior of Byzantine node.

Overall, the use of double-pair programming will allow us to address the challenges posed by this project. By dividing the work between two pairs of individuals, we can focus our efforts on specific areas of the project and ensure that each aspect receives the attention it requires. By doing so, we are confident that we can successfully implement the Raft consensus algorithm and create a robust, reliable distributed system.

9 Related Work

Copeland and Zhong [4] present a new algorithm called Tangaroa in Haskell, which extends the Raft consensus protocol to provide Byzantine fault tolerance. It does so by adding an additional round of communication between the leader and followers to detect and exclude misbehaving nodes. The paper shows that Tangaroa is efficient, can tolerate a large number of Byzantine faults, and maintains high throughput and low latency. It incorporates some features from the PBFT algorithm (including signatures, malicious leader detection, and election verification) into the Raft algorithm to achieve Byzantine fault tolerance while maintaining feasibility and robustness. This paper also inspired projects like Kadena to develop better-performing Byzantine fault-tolerant algorithms.

Yeh et al. [3] propose a solution that uses signed message transmission and forwarding of the leader's message to all other followers to ensure consistency. This enhancement involves some overhead, as each time the leader sends a command to followers, a total number of $O(N^2)$ messages must be forwarded across the cluster, where N represents the number of hosts in the cluster. Despite this, the enhanced RAFT protocol is able to maintain consistency and is more resilient to Byzantine faults, making it a valuable solution for distributed systems that require fault tolerance.

Clow and Jiang [5] come up with two version of BFT-Raft implement in Python and Rust with digital signatures, cryptographic digests with slightly fewer performance optimizations but is easier to understand and implement correctly.

Yossi Gilad and Silvio Micali from MIT's Computer Science and Artificial Intelligence Laboratory (CSAIL) addressed the problem of poor performance of the PBFT algorithm in many-node scenarios [6]. They proposed selecting a small number of accounting nodes first, and then using a Verifiable Random Function (VRF) to randomly select a leader node, avoiding direct consensus across the entire network. This extended Byzantine fault tolerance to support larger-scale applications while maintaining good performance.

10 Future Work

10.1 Pre-Append Phase

Under the assumption of an honest leader, splitting the append phase into pre-append and append could further improve the Byzantine fault-tolerance of the system. The design of this method is to use the pre-append message as authentication and coordination and use the append phase to transfer actual data with low priority, thus improving the overall throughput. During the pre-append phase, the leader packs all the transactions into a single packet, with the current term, the hash of transactions (for authentication), and the leader's signature. Combining the hash and the signature ensures that no malicious could alter the content or create false transactions that could stall the system. When a replica receives the pre-append packet, it first verifies the leader term and signature. Then it compares the combination of the entry number and the term, ensuring that the leader is at least or more up-to-date. The replica node adds the pre-append packet to its log if the checks pass, and respond to the leader with the term, log numbers, and also the signatures requesting the append packet, which contains the actual data. Using the hash of the transaction the leader and followers could ensure the system integrity, but at the cost of performance since it resembles the idea of a block-chain. Once the two phases are completed, the follower sends the signature, commit number, and the hashes after append to finish the commit process.

10.2 Commit verification

In addition to the mechanisms we implemented, there are some other methods that could further increase the robustness and fault tolerance of the BFT-Raft. One such feature is committed verification. In the system, we must assume that non of the messages are secure so we must treat them carefully with a series of verification and authentication steps. Commit verification requires each server to broadcast AppendEntriesResponse to other nodes in the system. The non-BFT version only sends to response to the leader, which could cause some inconsistency if the leader has been compromised. Furthermore, each replicated server also has the power to increment the commit index. It can be achieved by storing the

previous AppendEntriesResponse RPCs received from other nodes. Based on the state of the system and information fields in the RPC, the node can determine whether to accept the new commit index or discard it. This essentially grants the replica node the power to dictate its own state and restrain some of the leader's power. Combining the incremental hash technique of the log, this method converts the raft consensus protocol into a more decentralized, blockchain-like protocol, further improving the security of the system.

11 Conclusion

In this paper, we propose an implementation of the Raft consensus algorithm that uses gRPC as the communication protocol between replicas and clients. We extend this implementation to provide Byzantine fault tolerance, enabling the system to withstand arbitrary faults including malicious attacks. We adopted several methods to achieve BFT: message signature, client intervention, incremental hashing, election verification, and lazy voters. We have the assumption that adding those features would have negative impacts on the overall performance, and the benchmark result showed that our assumption is consistent. In the application setting, the user could choose to apply different sets of security features to fine-tune the balance between speed and security.

12 Code

We implement all code from scratch, and you can find our Python implementation at:
https://github.com/silongtan/BFT-Raft

Basic Raft implementation is at another branch:
https://github.com/silongtan/BFT-Raft/tree/basic_raft

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[2] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," pp. 382–401. [Online]. Available: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/

[3] T.-C. Yeh, S. He, Y. Zhang, and Y.-C. Lin, "Project report byzantine fault tolerant raft." [Online]. Available: https://www.scs.stanford.edu/17au-cs244b/labs/projects/clow_jiang.pdf

[4] C. Copeland and H. Zhong, "Tangaroa: a byzantine fault tolerant raft." [Online]. Available: https://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf

[5] J. Clow and Z. Jiang, "A byzantine fault tolerant raft." [Online]. Available: https://www.scs.stanford.edu/17au-cs244b/labs/projects/clow_jiang.pdf

[6] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, pp. 51–68. [Online]. Available: https://dl.acm.org/doi/10.1145/3132747.3132757