# CS6858 - Consensus Algorithm - Project Report

Rishabh Singh Gaharwar CS21B067
Ayman Akhter CS21B012

April 2024

## 1 Introduction

Byzantine Fault Tolerant algorithms are widely popular and find uses in many applications with high security demand. Systems based on BFT algorithms are said to be *intrusion-tolerant*. *State Machine Replication* is one of the techniques majorly used in these algorithms. In this form of replication, all honest servers have to execute all clients' requests in the same order.

Several *leader-based* Byantine fault-tolerant State machine replication algorithms are known, of which, PBFT is the most popular. In such algorithms, servers move through a succession of *views*. Each view has a *primary* or *leader*, that is in charge of defining the order in which requests are executed by the servers. It is interesting to note that in PBFT, the primary stays the same unless a view change is initiated.

This scheme is vulnerable to *performance attacks* [1], which make it so slow that it is barely usable. There are two well-known attacks

1. *pre-prepare delay*: A faulty server delays the ordering of requests from some clients, reducing throughput. Although PBFT imposes a maximum delay on execution of requests, a faulty primary can execute one request at a time, delaying future requests.

2. *timeout manipulation*: Faulty servers manage to increase the timeouts used in PBFT, degrading the performance of the system.

This project focuses on *Spinning BFT* : A novel algorithm, that modifies usual form of PBFT. The primary is changed *whenever the primary defines the order of a single batch of requests.* Spinning BFT has the three communication steps of PBFT, but there is no view-change method, since views are changing in every round. Instead, there is a *merge* operation, which is incharge of ensuring correctness and consistency across all honest servers.

Servers are also blacklisted if found to be faulty and once a server is blacklisted, it can never become the primary in future. Spinning has two major benefits

1. Avoids above-mentioned attacks made by faulty primary, since primary is always changed.

2. Improves the throughput of PBFT in fault-free case, by balancing load of ordering requests among all servers (by around 20%)

# 2 System Model

The system is composed of a set of $n$ servers $\Pi = \{s_0, \ldots, s_{n-1}\}$ that provide a Byzantine Fault Tolerant service to a set of clients $C = \{c_0, c_1, \ldots\}$. Clients and servers are interconnected by a network and communicate only by message passing.

A partial synchrony model is assumed in the paper, i.e., in all executions of the system, there is a bound $\Delta$ and an instant `GST` so that every message sent by a honest server to another honest server at instance $t >$ `GST` is received before $t + \Delta$ with $\Delta$ and `GST` unknown. This assumption is required to ensure liveness of BFT algorithm.

Any number of clients can be Byzantine, but the numbers of servers that can be faulty is limited to $f = \lfloor \frac{n-1}{3} \rfloor$ and thus $n \geq 3f + 1$. For simplicity, henceforth $n = 3f + 1$.

Communication is done using point-to-point authenticated channels, and a sequence of messages may be lost. Authenticity of messages exchanged is protected with signatures based on public-key cryptography or message-authentication-code (MACs) produced with collision resistant hash functions.

# 3 Algorithm

During the execution of the algorithm, each server maintains a set of *state variables* that are modified by a set of *operations*. Clients issue *requests* with operations.

The properties enforced by this algorithm are

1. **Safety**: All correct servers execute the same requests in the same order.

2. **Liveness**: All correct clients' requests are eventually executed.

Each server is always in one of two states:

1. `normal` state.

2. `merge` state.

Considering the description of the algorithm in `normal` state first:

1. *Client sends request to all servers*:

   A client $c$ issues a request for operation $op$ by sending a message $\langle \texttt{REQUEST}, c, seq, op \rangle_{\sigma_c}$ to all servers. $seq$ plays the role of client-request identifier. Servers do not execute a request for a client with a $seq$ lower than the last executed of that client.

2. *Server $s_i$ necomes primary of view $v$*:

   When server $s_i$ becomes the primary of view $v$ it first verifies the following conditions

   (a) It accepted the request from view $v - 1$

   (b) It is in `normal` state

   (c) If it has atleast one client request pending to be ordered.

   If the conditions are satisfied, primary sends $\langle \texttt{PRE-PREPARE}, s_i, v, dm \rangle_{\sigma_{s_i}}$ message to all servers, where $dm$ is a digest of request sent by client.

3. *Server $s_j$ receives a `PRE-PREPARE` message*

   On receiving $\langle \texttt{PRE-PREPARE}, s_i, v, dm \rangle_{\sigma_{s_i}}$ message from $s_i$, it evaluates if

   (a) The signature is valid

   (b) The view number $v$ is equal to the current view number on $s_j$ and sender is primary of view $v$

   (c) It is in `normal` state

   If conditions are satisfied, the `PRE-PREPARE` message is *valid* and $s_j$ sends a $\langle \texttt{PREPARE}, v, s_j, dm \rangle_{\sigma_{s_j}}$ message to all other servers. It discards any other `PRE-PREPARE` message with view $v$ [1].

4. *Server $s_j$ receives `PREPARE` messages from atleast $2f + 1$ servers*

   When $s_j$ has $2f + 1$ valid `PREPARE` messages (including its own), it sends a $\langle \texttt{COMMIT}, v, s_j \rangle_{\sigma_{s_j}}$ message to all servers.

5. *Server $s_j$ receives atleast $2f + 1$ valid `COMMIT` messages*

   The request is accepted and executed. After executing the request, the server sends $\langle \texttt{REPLY}, s, seq, res \rangle_{\sigma_{s_j}}$ to the client that issued the request. The view number of the server is incremented.

   - If $s_j$ becomes the primary in the new view, it sends `PRE-PREPARE` message to other servers
   - Otherwise, $s_j$ sets a timer waiting for the request of new view

   If client receives $f + 1$ replies with same result $res$ from different servers, it accepts the result.

[1] Note that $s_j$ can also receive `PRE-PREPARE` message with view $v' > v$. In this case, such messages are stored in the buffer until view $v'$ is reached.

Now we describe the algorithm in *merge* state

1. *Upon timer expiration on server $s_i$:*
   - Server $s_i$ sends a $\langle$MERGE, $s_i$, $v$, $P\rangle\sigma_{s_i}$ message to all servers.
   - If $s_i$ has at least $f+1$ PREPARE messages with the current view number, $v$ field is set to this view number, else it's set to $v_{last}$.
   - $P$ contains prepare certificates for requests from the current and previous views that $s_i$ accepted.
   - After sending MERGE, $s_i$ changes its state to merge and increments the view number.
2. *Upon server $s_j$ receiving $f+1$ MERGE messages:*
   - If $s_j$ receives $f+1$ MERGE messages for a view $v$, with $v$ higher or equal to its current view and it hasn't sent a MERGE message for this view, it sends $\langle$MERGE, $s_j$, $v$, $P\rangle\sigma_{s_j}$ to all servers.
   - Server $s_j$ changes its state to merge and increments the view number.
3. *Upon server $s_i$ becoming the primary of a new view $v$:*
   - If $s_i$ receives $2f+1$ MERGE messages for view $v-1$ and becomes the primary of the new view $v$, it sends a $\langle$PRE-PREPARE-MERGE, $s_i$, $v$, $VP$, $M\rangle\sigma_{s_i}$ message to all servers.
   - $VP$ is a vector of digests of requests taken from the $P$ field of the MERGE messages, ordered by view number.
   - $M$ is a merge certificate composed of the $2f+1$ MERGE messages received.
4. *Upon server $s_j$ receiving a PRE-PREPARE-MERGE message:*
   - If server $s_j$ receives a valid $\langle$PRE-PREPARE-MERGE, $s_i$, $v$, $VP$, $M\rangle\sigma_{s_j}$ message from $s_i$:
     (a) It verifies the signature, sender, and $VP$'s validity using merge certificate $M$.
     (b) If $v_{last}+1 \geq v_{min}$, server $s_j$ changes its state to normal and sends a $\langle$PREPARE, $v$, $s_j$, $dm\rangle\sigma_{s_j}$ message to all servers, where $dm$ is the digest of $VP$.
     (c) Server $s_j$ then follows steps 4 and 5 of normal operation, avoiding re-executing clients' requests using previously executed requests' information.
     (d) Otherwise, if $s_j$ missed some messages before the view $v_{min}$, it obtains missing information from another server using *state transfer mechanism.*

# 4 Implementation Details

## 4.1 Programming Languages and Frameworks

The implementation is done in JavaScript, utilizing the Node.js runtime environment. Node.js provides an event-driven, non-blocking I/O model, which is well-suited for building networked applications.

## 4.2 Architecture Overview

The application is structured as a distributed system, where multiple servers communicate with each other over a network using TCP sockets. Each server maintains connections to its peers and can send and receive messages in the form of JSON-encoded tuples. The main components of the system include:

- **Server**: Represents an individual node in the distributed system. Each server listens for incoming connections and handles various types of messages.
- **Connection Manager**: Manages connections to peer servers and facilitates message exchange between servers.
- **Message Handlers**: Functions responsible for processing different types of messages received from peers.
- **State Management**: Tracks the state of the server, including view number, message history, pending messages, and processing state.

## 4.3 Error Handling and Recovery

The code includes error handling mechanisms to deal with various error conditions, such as network errors, connection failures, and invalid messages. Error messages are logged to the console for debugging purposes, and appropriate actions are taken to recover from errors and maintain system stability.

# 5 Fault Tolerance in Implementation

In the Spinning BFT consensus algorithm, the MERGE mechanism is used to tolerate Byzantine faults, specifically when the faulty primary may not send the PRE-PREPARE message or only sends it to some of the servers, leading to inconsistencies in the ordering of client requests.

**Timeout for Acceptance ($T_{acc}$):**

- When a server has client requests in its buffer to be ordered, it waits for a maximum time interval $T_{acc}$ to accept the request of that view.
- If the server does not receive enough COMMIT messages to accept the request during $T_{acc}$, it sends a MERGE message to all servers.

- In the paper, it is mentioned that $T_{acc}$ is multiplied by 2 in every merge operation, to ensure the liveness of the system.
- However, to avoid malicious primary influencing this timeout to be high and the progress of the system to be slow, each server divides by two the value of $T_{acc}$ whenever system it stable.
- To determine if the system is stable, every primary calculates the average time taken to execute a request. If it is smaller than $T_{acc}\overline{_{2,then}}\mathrm{T}_{acc}$ is divided by 2.

**MERGE Operation:**

- The MERGE operation ensures liveness if the primary is faulty.
- It ensures that all correct servers agree on which requests of the previous views were accepted and need to be executed.
- When a server $s_i$ receives at least $f + 1$ MERGE messages, it sends its own MERGE message and changes its state to *merge*.
- When the server that will be the primary in the next view receives $2f + 1$ MERGE messages, it sends a PRE-PREPARE-MERGE message.
- The PRE-PREPARE-MERGE message carries enough information to make all servers agree on the order of requests in the previous views.
- When another correct server receives and validates the PRE-PREPARE-MERGE message, it sends a PREPARE message and follows the normal operation of the algorithm.

This function `MergeHandler` handles the MERGE message. When this message is received, it increments the merge_count. If merge_count is greater than or equal to $2f + 1$, it sets the state to 'merge', resets some flags (`isPrePrepared` and `isPrepared`), and then iterates over the processing map to find the `myP` value. Afterward, it sends the MERGE message to all servers.

To dive into the details:

1. Increment `merge_count`.
2. Check if `merge_count` is greater than or equal to $2f + 1$.
3. If the condition is met, set the state to 'merge', reset some flags (`isPrePrepared` and `isPrepared`).
4. Iterate over the processing map:
   (a) If a valid entry is found (`value.length >= 2 * f + 1` and `key[1] >= v_last - n`), set `myP`.
   (b) Send the MERGE message to all servers.

(c) Exit the loop.

The MERGE message contains the `my_view`, `host_id`, and `myP`. All these details are used during stateTransfer if that is needed.

Also, when a server received sufficient number of MERGE messages for a view $v$, it knows that the primary of view $v-1$ may have been faulty. Thus, that server is added to a blacklist.

# 6 Results

## 6.1 Fault Free Case

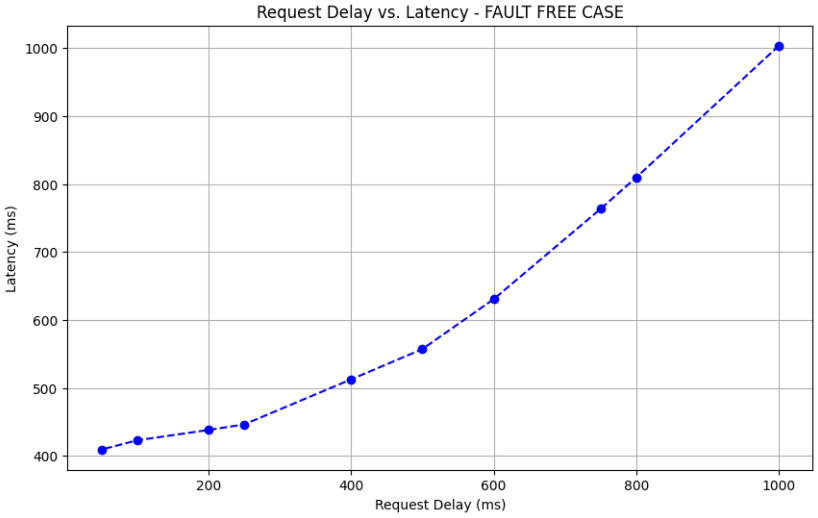


Figure 1: Latencies (f=0)

As the processing delay of requests decreases, the benefits in terms of reducing latency also diminish. Initially, reducing the delay significantly impacts latency reduction. However, as the delay approaches lower values, the improvements in latency become less pronounced. This suggests that there is a point beyond which further reductions in delay may result in only marginal improvements in latency.

The fault free case is also the instance where Spinning BFT shines. Spin BFT has higher throughput than PBFT when there are no faults.

## 6.2 Crash Failure

Flat line in the plot shows the revive time after crash. The system goes into merge state after detecting a crash, and after that continues to process requests.

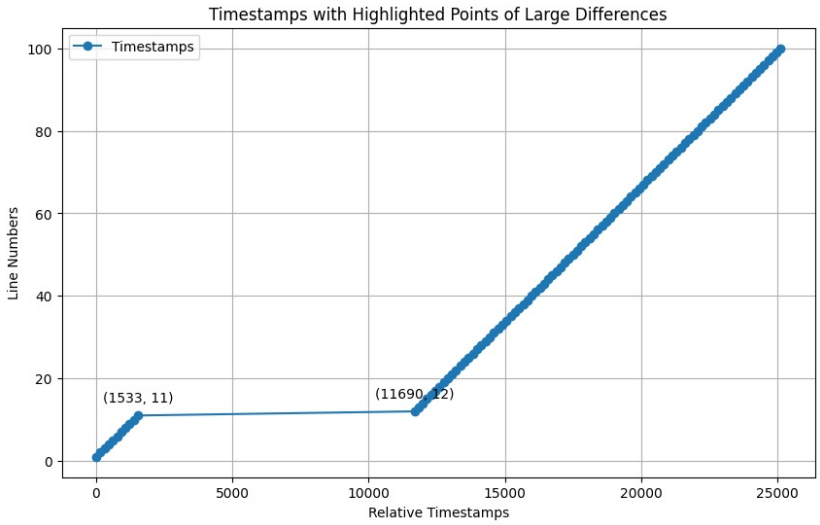


Figure 2: Request Processing vs time for crash failure

In simpler terms, as we reduce the delay in processing requests, the resulting reduction in latency becomes less significant. This implies that there is a threshold beyond which additional decreases in delay may offer only minimal benefits in terms of improving overall latency.

### 6.3 Byzantine Failure

In this implementation, byzantine primaries are simulated using two methods

- Byzantine primaries do not send any messages to any servers.
- Byzantine primaries send messages to servers selectively.

The first case is already handled in crash failure case. The second case is handled using the `merge` operation. In the merge operation, when a server's timer expires, the server sends a merge request, and rest of the algorithm is followed.

## 7 Conclusion

Spinning Byzantine Fault Tolerance (BFT) is a consensus algorithm designed to ensure the security and reliability of distributed systems, particularly in scenarios where some nodes may be malicious or faulty. Unlike traditional BFT algorithms that use a leader-based approach, Spinning BFT employs a decentralized approach, eliminating the need for a designated leader. Instead, it relies on a rotating leadership model, where each node takes turns proposing blocks

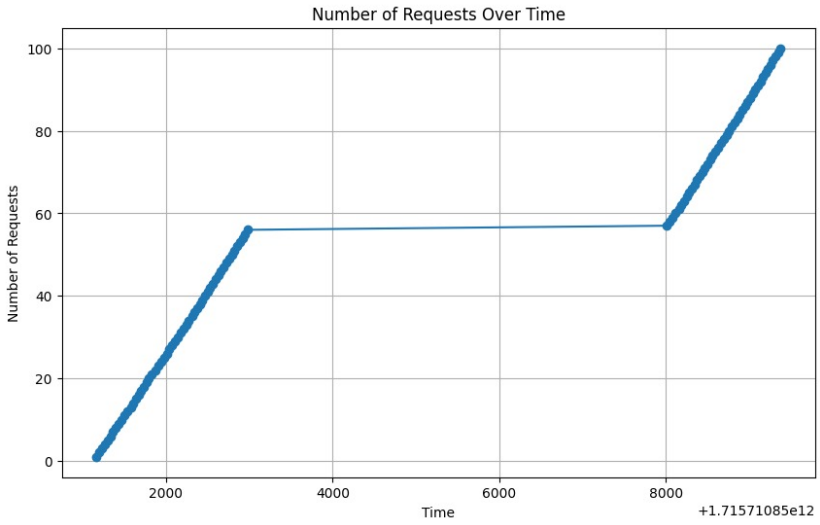


Figure 3: Request Processing vs time for byzantine failure

and reaching a consensus on the order of transactions. This ensures that no single point of failure exists, enhancing the system's fault tolerance and resilience.

One of the key advantages of Spinning BFT is its ability to achieve high throughput and low latency in large-scale distributed systems. By eliminating the bottleneck caused by a single leader, Spinning BFT distributes the responsibility of block proposal among all nodes, allowing for parallel transaction processing. Additionally, its decentralized nature makes it highly resilient to various attacks, including Sybil attacks and DDoS attacks. However, Spinning BFT requires careful tuning of parameters to balance between safety and liveness, and it may face challenges in networks with high churn rates. Overall, Spinning BFT represents a significant advancement in the field of distributed consensus, offering a robust and efficient solution for building secure and reliable decentralized applications.

# 8 Scope of Improvement in Implementation

The implementation of Spinning Byzantine Fault Tolerance (BFT) could be significantly enhanced by integrating advanced cryptographic schemes and a robust Public Key Infrastructure (PKI). By leveraging digital signatures and encryption techniques, the protocol can ensure message authenticity, integrity, and confidentiality, thus mitigating the risk of various malicious attacks. Incorporating a PKI would further strengthen the network's security by authenticating the identities of participating nodes, preventing Sybil and Man-in-the-Middle attacks, and securely distributing public keys.

Furthermore, to improve scalability, the Spinning BFT protocol could benefit

from optimizing the block proposal and validation process. Merge and State Transfer can be made more efficient and robust with dynamic updation of $T_{acc}$ which would dynamically change the timeout duration of every server.

# References

[1] Giuliana Santos Veronese et al. "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary". In: *2009 28th IEEE International Symposium on Reliable Distributed Systems*. 2009, pp. 135–144. DOI: 10.1109/SRDS.2009.36.