*Original scientific paper*
*Received: 2022-03-22*
*Accepted: 2022-04-06*

# BYZANTINE FAULT TOLERANT RAFT ALGORITHM WITH ROUND ROBIN LEADER ELECTION

Mateo LUČIĆ, RIT Croatia, mxl3773@g.rit.edu

**Abstract**

To implement a Byzantine fault tolerant Raft algorithm, a variant of the Raft algorithm with modified leader election and log replication is proposed. This modification requires that all messages between nodes are cryptographically signed, and that each transaction requires that the leader provides proof of valid quorum. The original Raft leadership election is modified via the round robin algorithm to prevent Byzantine nodes from obstructing the normal behavior of the consensus cluster.

**Keywords**: Raft Algorithm, Byzantine fault tolerance

## 1. Introduction

More and more of the software nowadays is being migrated to the various cloud offerings and is moving to a more decentralized model of use. This shift to a more decentralized model also extends to various methods of data storage ensuring that the data is always available. This data is either sharded or replicated (sometimes even both) and is distributed on the storage mediums that are not necessarily located within the same server rack. This distribution is therefore highly sensitive to any disruptions on the storage mediums and must be made as resilient as possible.

Synchronization of the storage mediums is generally controlled and managed by an implementation of a consensus algorithm such as Paxos (Pease et al., 1980 ) or Raft (Ongaro & Ousterhout, n.d.), but those algorithms are not Byzantine Fault Tolerant.

The Byzantine Generals Problem (Lamport et al., 1980) is a specific occurrence in the digital systems (especially distributed ones) in which a component might fail in a way that still registers to the system that it is operational, therefore presenting a misleading health indicator to the rest of the system.

While the Raft algorithm provides a level of resilience in case of follower failure, it does not adequately handle a possible Byzantine leader, nor can it properly handle a Byzantine follower. Therefore, this paper will discuss a Byzantine fault tolerant adaptation of Raft algorithm with a particular focus on hardening the leader election and log replication.

## 2. Background

### 2.1. Related Work

There are multiple approaches to providing the resistance to Byzantine Faults, such as using symmetric key cryptography (Ateniese et al., 2008) and message signing (Tian et al., 2021), using document hash chains and generated timestamps (Abadi et al., 2020). Another promising avenue is to use the blockchain principles in order to detect failures in the system when verifying the chain (Narayanan, 2016)

Unfortunately, adding additional system checks to an already existing algorithm will reduce the performance of these algorithms. This fact therefore necessitates that the availability check is performed in the least possible amount of steps. A good method to implement said functionality would be to remove expensive Byzantine signatures and to set the minimum of $5\boldsymbol{f} + 1$ acceptors which will allow it to have up to $\boldsymbol{f}$ faults, $3\boldsymbol{f} + 1$ proposers and $3\boldsymbol{f} + 1$ (Martin & Alvisi, 2006).

While the Paxos consensus algorithm is a much more mature algorithm than Raft, it has already been a subject of similar research. An example of said research would be the Cheap Paxos implementation by Leslie Lamport and Mike Massa, which can handle faults up to 2F + 1 processors, but does such on the assumption that the non faulty processors do not jump around as fast (Lamport & Massa, 2004).

A similar modification was proposed by (Dumovic & Jain, n.d.) called Battleship. While it shares the general approach to the problem of the Byzantine fault tolerance as this proposal, it introduces several complexities that can be further simplified.

### 2.2. Statement of the Problem

Raft is a consensus algorithm which produces an equivalent result to multi-Paxos, but with the additional goal of being simpler and more understandable. This is achieved by separating the key elements of the consensus (leader election, log replication), and reducing the number of states that the state machine needs to consider.

Raft algorithm guarantees that the following properties are always true:

- Election Safety
- Leader Append-Only
- Log Matching
- Leader Completeness
- State Machine Safety

While the existence of a strong leader simplifies the management of the replicated log, and the leader election allows a certain level of fault resilience, it also introduces a dangerous failure point for malicious actors.

## 3. Failure Points

The failure points can be roughly divided into the following categories: Client - Cluster fault, Consensus fault and Leadership election fault.

Client-Cluster faults occur when the leader is either not responding or falsifying client requests or responses.

Consensus faults are caused by the faulty node, which can either attempt to respond twice with conflicting results, or may attempt to overwrite existing commits, or attempting to impersonate other nodes.

Leadership Election faults are caused by faulty nodes attempting to rig the election by delaying votes of other candidates, attempting to vote multiple times or by falsely initiating the voting in order to lock up the cluster.

## 4. Providing Byzantine Fault Tolerance

### 4.1. Assumptions and Necessary Configurations

The assumptions for this protocol are as follows:

- The client is always sending valid commands and is therefore trusted
- The malicious actor does not have control over the network, therefore it cannot interfere with the messages from non faulty nodes.
- Client is aware of all nodes in the cluster
- Each node has an integer identifier that is incremental, and starts from 0

To provide the Byzantine fault tolerance, the cluster needs to form a valid quorum in face of $f$ failures. Since the minimum number of replicas in a Byzantine fault tolerant system is $3f + 1$, a valid quorum needs to contain $2f + 1$ valid responses.

To prevent forging of the node to leader communication, and in order for the client to validate the cluster response, the messages must be cryptographically secure. This is achieved by generating a set of public RSA keys that are shared across nodes and clients, while the associated private keys are secret and known only to the issuing node. The nodes therefore are able to verify that the message received is actually from the sending node instead of being spoofed.

To prevent a possible byzantine leader from forging the client commands, the commands themselves are also encrypted with the RSA client key. The client public key is also known to all nodes.

Unless otherwise specified in the next section, the behavior of the cluster itself is the same as in the original Raft specification.

In the following section, all messages that cannot be verified via the public key associated with the originating node will be considered as invalid and will be ignored.

## 5. Implementation

### 5.1. Log Replication

In order to properly append the new data to the log, the leader needs to ensure that the log entry in question is prepared on $2f + 1$ nodes in order to ensure the existence of data. The original algorithm performs this operation in one step with a single *AppendEntry* RPC call, but this behavior does not ensure eventual existence of the data in question.

This necessitates that the *AppendEntry* RPC call is split into two RPC calls:

- *PrepareEntry* RPC
- *CommitEntry* RPC

Both the *PrepareEntry* and *CommitEntry* messages are encrypted with the leader private key, while the acknowledgements to the same are encrypted with the originating node private key. Additionally, the *PrepareEntry* RPC contains the original command sent by the client in order to ensure no tampering occured.

The *PrepareEntry* RPC call sends the data to the follower nodes. The follower nodes validate that the command originated from the client and that it was not tampered with by decrypting the client command. In case the command is confirmed valid with the client public key, that data is then prepared for commit, and the nodes notify the leader as soon as the message is prepared with the generated hash for said update.

Once the leader receives $2f + 1$ acknowledgements to the *PrepareEntry* RPC, it constructs a *CommitEntry* RPC message that needs to contain the proof of $2f + 1$ acknowledgements that the leader received for the node to commit the log entry. Additionally, all the acknowledgements need to contain the same message hash in order to ensure that the leader did not send different messages to different nodes. If the *CommitEntry* RPC message does not contain $2f + 1$ acknowledgements, or if the hashes provided in the acknowledgements do not match the prepared hash and leader provided hash, the message is ignored.

As with the *PrepareEntry* RPC, the follower nodes will return acknowledgement of the commit to the leader, which are passed to the client as a proof of replication, in addition to the result of the operator request.

If the client does not receive the response in a reasonable timeframe, or if the proofs contained within the same response do not match each other, the client can trigger leader re-election by broadcasting *ProposeTermChange* RPC to all nodes in the cluster.

## 6. Leader election

In the original Raft algorithm, leader election is triggered when one of the following conditions are met:

- The leader stopped sending heartbeat messages
- The election timer timed out on one of the nodes

While this system works well in theory, in reality it is susceptible to faults caused by either accidents or malicious behavior.

Therefore, to harden the leader election, the original leader election is replaced with a round robin algorithm. The round robin algorithm is triggered when one of the aforementioned conditions (no heartbeat messages from the leader or election timeout) are met and selects a new leader based on the *t mod n* formula, in which *t* represents the next term, and *n* represents the number of nodes in the cluster.

Once the candidate node determines which node should be the new leader, it broadcasts the message via the *ProposeTermChange* RPC that contains the index of next turn and calculated ID for the next leader. After sending the *ProposeTermChange* RPC, the sender resets the election and timeout timers.

If the calculated leader is invalid according to the *t mod n* formula, or if the next term is not the next term index as written in the node state, the message is discarded.

Once the next leader receives the message, it sends *TermChange* RPC containing the hash of the last committed log entry. If said log entry does not match the last log entry on the receiving node, the message is discarded. If the last committed log entry hash is valid, the node signals its acceptance of the leader.

Once the proposed leader acquires the $2f + 1$ acceptances from other nodes, it commits the term change via the *Log Replication* methodology as previously described, with the acceptances being used as a proof of validity.

Each node logs the *ProposeTermChange* RPC that triggers this process in order to handle a Byzantine perspective leader. In case a *ProposeTermChange* RPC is triggered, but no leader was elected within its own election or heartbeat timeout timer, it increments the term in *ProposeTermChange* RPC and triggers a new *ProposeTermChange* RPC broadcast with the new increment. This procedure is repeated until a new leader is elected, or until the client triggers reelection.

## 7. Evaluation

Compared to the normal Raft algorithm implementation, the proposed modification is significantly slower. During the testing, it has been noticed that there is a minor increase in time required for the cluster to commit the new log due to usage of RSA encryption and modified *log commit* and *leader election* protocols.

The testing covered all the potential failure points as specified previously, and this implementation managed to provide tolerance to said failure points. The following tests were performed on 4- and 7- member clusters.

Client cluster failure tests consisted of two cases - cluster not responding to the requests and cluster returned tampered response. In the case of the cluster not responding to the requests, the cluster contained one leader that refused all requests after the election, but would still send heartbeats in order to postpone election. Once the client request timeout expired (600 ms), it broadcasted

*ProposeTermChange* RPC to the cluster to trigger election. Said election replaced the byzantine leader and the request was resent and properly processed. The behavior of the cluster was similar in case the leader returned a tampered response. Once the client noticed that the signatures in the proof of work do not match each other, it triggered *ProposeTermChange* RPC and performed elections.

Consensus error failure tests consisted of two cases:

- Byzantine node attempts impersonation of another node
- Byzantine node sends two conflicting responses to an RPC

The node attempting to impersonate another node used its own private key to sign the message, while providing the identifier of another node. When the leader attempted to decode the message using the public key associated with the impersonated node, the decryption failed and said proof of work was disregarded.

When the byzantine node attempted to send conflicting messages, the leader node would disregard any messages that did not match the expected commit hash. This behavior was expected and noticed in both *PrepareEntry* and *CommitEntry* RPCs.

Leader Election fault testing comprised of four test cases:

- The next leader does not respond to *ProposeTermChange* RPC
- Byzantine nodes collude to switch terms between themselves by proposing invalid term identifiers.
- Byzantine node acknowledges leader election multiple times
- Byzantine node triggers elections prematurely

While the next leader did not respond to *ProposeTermChange* RPC within the election timer of the nodes, the broadcasts itself were logged by receiving nodes. Since the leader election is triggered by either election or heartbeat timeout, the next node that triggered any of the timeouts increased the term identifier and sent a new *ProposeTermChange* RPC with the new timer. The node associated with that identifier responded to the *ProposeTermChange* and the election procedure continued normally.

When two byzantine nodes attempted to propose term change to a term that is not next in the sequence, the proposed leader did not send *TermChange* RPC since it can internally verify that it is not yet the requested term (id not next in sequence, there was no non responsive *ProposeTermChange* broadcasts). In one case that the term proposed by the byzantine node was another byzantine node, and said node did send out *TermChange* RPC, the other nodes did not respond with acceptance due to the internal term verification.

When the node was set to trigger elections every 10 - 23 ms (instead of standard 150 - 300 ms), the *ProposeTermChange* requests would be ignored by the next leader since it could verify that the time elapsed from the previous term change was less than the lower bound for election timeout. Attempts to send multiple acknowledgements to the election were discarded, since the proofs of approval need to be unique per node.

## 8. Conclusion

This modification to the Raft consensus algorithm implements Byzantine fault tolerance by enforcing cryptographic signing of messages in order to prevent node impersonation and requiring proof of validity for leadership elections and log commits in order to prevent Byzantine leaders and nodes.

Additionally, it removes the possibility of a Byzantine leader preventing progress in the system by implementing a round robin leadership change method that does not depend on trust in the cluster.

## References

Copeland, C., & Zhong, H. (n.d.). *Tangaroa: a Byzantine Fault Tolerant Raft*. Stanford Secure Computer Systems Group. Retrieved March 1, 2022, from https://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf

Dumovic, M., & Jain, S. (n.d.). *Battleship: Byzantine Fault Tolerant Raft*. Stanford Secure Computer Systems Group. Retrieved March 1, 2022, from http://www.scs.stanford.edu/17au-cs244b/labs/projects/dumovic_jain.pdf

Lamport, L., & Massa, M. (2004, June). *Cheap paxos* [Paper presentation]. Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004), Florence, Italy.

Lamport, L., Shostak, R., & Pease, M. (1982, July). The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, *4*(3), 382 - 401. ACM Transactions on Programming Languages and Systems

Martin, J.-P., & Alvisi, L. (2006). Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, *3*(3), 202-215. https://doi.org/10.1109%2FTDSC.2006.35

Ongaro, D., & Ousterhout, J. (2014, May 20). *In Search of an Understandable Consensus Algorithm (Extended Version)*. Raft Consensus Algorithm. Retrieved March 1, 2022, from https://raft.github.io/raft.pdf

Tian, S., Zhang, Y., Liu, Y., & Zhao, Y. (2021). A Byzantine Fault-Tolerant Raft Algorithm Combined with Schnorr Signature. *International Conference on Ubiquitous Information Management and Communication (IMCOM)*. 10.1109/IMCOM51814.2021.9377376