Tangaroa: a Byzantine Fault Tolerant Raft

Christopher Copeland and Hongxia Zhong

***Abstract*— We propose a Byzantine Fault Tolerant variant of the Raft consensus algorithm, BFTRaft, inspired by the original Raft[1] algorithm and the Practical Byzantine Fault Tolerance algorithm[2]. BFT Raft maintains the safety, fault tolerance, and liveness properties of Raft in the presence of Byzantine faults, while also aiming towards to Raft's goal of simplicity and understandability. We have implemented a proof-of-concept of this algorithm in the Haskell programming language.**

I. Introduction

The Raft consensus algorithm is in many ways superior to Paxos and other consensus algorithms. In designing Raft, Ongaro and Ousterhout applied specific techniques to improve understandability, including decomposition and state space reduction. The simplicity of Raft leads to a number of unique advantages for both educational purposes and as a foundation for implementation. In order to keep the Raft algorithm simple and understandable, Raft assumes that nodes fail only by stopping, which rarely holds in practice unfortunately. Malicious attacks and software errors can cause faulty nodes to exhibit Byzantine (i.e., arbitrary) behavior and consequently subvert the correctness and availability guarantees of the Raft algorithm. We aim to enhance the original Raft algorithm such that it becomes tolerant to Byzantine server behaviors.

A. Byzantine servers break Raft

Leader election and log replication together provide the safety guarantees of Raft. Raft guarantees the correctness and availability of the system even if any minority of the nodes in a cluster fail. In the presence of Byzantine nodes, however, Raft's safety and availability are compromised.

1) Leader election: In the Raft algorithm, one node is elected as leader before the system makes any progress in a term. If the leader fails, Raft advances the term and elects another leader. Since any node can trigger an election and terminate the current term at any moment, a Byzantine node can effortlessly starve the whole system by perpetual elections, and consequently subvert Raft's availability.

2) Log replication: Raft uses a strong form of leadership that allows the distinguished leader to take complete responsibilty for replicating log entries. Specifically, the Raft leader serves as a single point of contact between the client and the rest of the system: it interprets the client requests, instructs replicas to store and commit log entries, and finally responds with a result. A Byzantine leader could modify a client's request and violate correctness. A Byzantine leader could also instruct replicas to commit a log entry before it has been safely recorded by a quorum of replicas, causing a correctness violation if other nodes fail and the entry is never replicated on a quorum of nodes. Lastly, a Byzantine leader can trivially confuse clients by responding with arbitrary results or simply ignore all client communications. Because Raft does not prevent a Byzantine node from becoming a leader, the system could provide incorrect results or appear unavailable to clients if a Byzantine leader is ever elected. Any node that is up to date can become leader trivially by starting an election, even if it is still receiving heartbeats from the current leader.

These examples suggest that Byzantine nodes can sabotage the Raft algorithm in many ways. Moreover, even non-malicious Byzantine nodes can easily subvert the protocol if they do not handle RPCs in the expected way. We need a few modifications to the Raft protocol to handle such scenarios.

B. Features of BFT Raft

In designing BFT Raft, we applied similar techniques and decomposition in order to preserve the simplicity and understandability of Raft, but we use several modifications and additions that provide Byzantine fault tolerance.

1) Message signatures: BFT Raft uses digital signatures extensively to authenticate messages and verify their integrity. For example, the leader replicates client messages along with the client signatures. This prevents a Byzantine leader from modifying the message contents or forging messages. Client public keys are kept separate from replica public keys to enforce that only clients can send new valid commands, and only replicas can send valid Raft RPCs.

2) Client intervention: BFT Raft allows clients to interrupt the current leadership if it fails to make progress. This allows BFT Raft to prevent Byzantine leaders from starving the system.

3) Incremental hashing: Each replica in BFT Raft computes a cryptographic hash every time it appends a new entry to its log. The hash is computed over the previous hash and the newly appended log entry. A node can sign its last hash to prove that it has replicated the entirety of a log, and other servers can verify this quickly using the signature and the hash.

4) Election verification: Once a node has become leader, its first AppendEntries RPC to each other node will contain a quorum of RequestVoteResponse RPCs that it received in order to become leader (this is sent before the leader accepts any new entries from clients). Nodes first verify that the leader has actually won an election by counting and validating each of these RequestVoteResponses. Subsequent AppendEntries RPCs in the same term to the same node need not include these votes, but a node can ask for them in its AppendEntriesResponse if necessary. This will happen if

the replica restarted and no longer believes that the current leader won the election for that term.

5) Commit verification: In order to safely commit entries as they are replicated, each AppendEntriesResponse RPC is broadcast to each other node, rather than just to the leader. Further, each node decides for itself when to increment its commit index, rather than the leader. It does this by keeping track of the AppendEntriesResponse RPCs received from the other nodes, which are signed and which include the incremental hash of the last entry in that node's log. Once a node has received a quorum of matching AppendEntriesResponse RPCs from other nodes at a particular index, it considers that to be the new commit index, and discards any stored AppendEntriesResponse RPCs for previous indices.

This differs from Raft in that the leader no longer has any special responsibility of coordinating commits. AppendEntriesResponse RPCs become closer to the broadcast PREPARE message from PBFT. Each node can verify for itself that a quorum of nodes have prepared up to a particular index and have matching entries for that index and all previous indices by checking the incremental hash.

6) Lazy Voters: A node does not grant a vote to a candidate unless it believes the current leader is faulty. A node comes to believe that its leader is faulty if it does not receive an AppendEntries RPC within its own election timeout, or it receives an UpdateLeader RPC from a client for that leader. This prevents nodes that start unnecessary elections from gaining the requisite votes to become leader and starve the system.

While introducing these new rules and techniques, we try to keep the distinguishing features of Raft as much as possible. For example, BFT Raft also uses a relatively strong form of leadership such that new log entries only flow from the leader to replicas. BFT Raft also uses a randomized election timeout. The following section explains BFT Raft's features in more detail.

II. BFT Raft Algorithm

The BFT Raft algorithm decomposes the consensus problem into two relatively independent subproblems similar to Raft's decomposition: log replication (Section II.C), and leader election (Section II.D). We designed BFT Raft to offer the same safety guarantees as Raft, which are listed below:

Election safety: at most one leader can be elected in a given term.

Leader Append-Only: a non-Byzantine leader never overwrites or deletes entries in its log; it only appends new entries.

Log Matching: if two nodes have the same incremental hashes at the same index, then their logs are identical in all entries up through 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 node has applied a log entry at a given index to its state machine, no other non-Byzantine node will ever apply a different log entry for the same index.

A. BFT Raft basics

A BFT Raft cluster that tolerates f Byzantine failures must contain at least $n \geq 3f + 1$ nodes, where $n - f$ nodes form a quorum. BFT Raft is configured so that nodes and clients have the public keys of each other node and client ahead of time. BFT Raft nodes and clients always sign before sending messages and reject messages that do not include a valid signature. A public key cryptosystem such as RSA is sufficient for this purpose and prevents message forgeries. Replays are also not a problem because Raft RPCs are idempotent. For client commands, we use a monotonically increasing per-client identifier to detect duplicated messages.

In BFT Raft, each node is in one of the three states: leader, follower, or candidate. Similar to Raft, BFT Raft divides time into terms, which start with an election. The winner of the election serves as the leader for the rest of the term. Sometimes, an election will result in a split vote, and the term will end with no leader.

Similar to Raft, we designed BFT Raft to maintain a high level of coherency between nodes' views of the current term. In Raft, when a node receives an RPC from an outdated peer, it will respond with the current term number in order to update its peer. A Byzantine node can repeatedly trigger elections, but unlike in Raft, BFT Raft nodes only update their term number in one of three situations: (1) upon receiving an AppendEntriesRPC that contains a quorum of votes for the sender in a new term, (2) when responding to a RequestVote for a higher term, or (3) when becoming a candidate. Arbitrary RPCs with higher term numbers do not increment a node's term as they do in Raft.

BFT Raft nodes communicate using RPCs, and the consensus algorithm requires four types of RPCs:

AppendEntries RPC: initiated by leaders to replicate log entries, provide a heartbeat, and communicate a successful election.

RequestVote RPC: initiated by candidates during elections.

SendRequest RPC: initiated by clients to request the cluster to execute a command to its replicated state machines.

UpdateLeader RPC: initiated by clients to request change of leadership.

B. Incremental hashing

Each node stores a cryptographic hash for each log entry, called an incremental hash. To compute an incremental hash at index i, the node computes the hash of the incremental hash at index $i-1$ appended to the log entry at index i. The recursive nature of incremental hashing enables it to verify the integrety of all log entries up through index i. This gives BFT Raft a variant of the *log matching* property that is robust to Byzantine nodes that may re-order or drop entries from the log. When two nodes agree on an incremental hash at index i, they have identical log entries at index i and all entries prior to i. The integrity of an entire log can be checked efficiently, with a constant amount of work for each new entry.

C. Log replication

The BFT Raft cluster begins servicing client requests once a leader has been elected. A client issues a SendRequest

RPC to a node it thinks is the leader to execute a command. The SendRequest RPC also contains a signature and a unique identifier for the command (typically a timestamp). The signature guarantees the authenticity and integrity of the client request, preventing Byzantine nodes from forging client requests. The unique identifier prevents Byzantine nodes from duplicating existing client requests. The signature and unique identifier guarantee that each command will be replicated at most once by each non-Byzantine node.

As in Practical Byzantine Fault Tolerance, clients need to wait for $f + 1$ matching replies to each request before exposing that result to application logic. This ensures that at least one honest node decided that a particular result was safely replicated to a quorum and should be externalized. The client records all responses to the pending SendRequest RPC. Each time a client receives a response to the pending request, it resets the *progress timeout*. If a client has not received a response over a period of time called the *request timeout*, it resends the SendRequest RPC.

When a leader receives a SendRequest RPC, it sends a signed AppendEntries RPCs in parallel to each replica. A leader will include a quorum of signed votes to support its authority in the current term on the first AppendEntries RPC to each node in each term. Subsequent AppendEntries RPCs can succeed without it once the leader sees a successful reply from that node, indicating that the replica accepts that the current leader won the election for the current term.

When a node receives an AppendEntries RPC, it checks if it was from what it believes is the current leader for the current term. If this is not the case, it can reply with an "unconvinced" response, indicating that the leader should send its votes again.

Replication: As in Raft, the node will check that it has a matching log prefix, but using the incremental hash, rather than the term of the previous entry. It then checks the authenticity of each of the new entries for itself. If the node has a matching previous entry and the new entries are valid, the node will append new entries to its log and compute the incremental hash at each new index. It will then broadcast its AppendEntriesResponse to each other node, which contains the incremental hash at the last new index.

Commit: When a node receives an AppendEntriesResponse, it will save it if it is for an index higher than the node's current commit index. Once a node receives a quorum of matching AppendEntriesResponses for a particular index, it is safe to commit everything up to that log entry. The node can then apply the newly committed log entries to its state machine and send results to the client directly. Nodes also store the results of committed entries, so they can be retransmitted to a client when a duplicate command is received.

When a leader receives an AppendEntriesResponse, in addition to storing the commit information to eventually detect a quorum, the leader will check if the AppendEntries RPC was successful. The node could have responded that it needed proof of that leader's successful election, or that it did not have the previous entry to the new entries. If the former, the leader will send a new AppendEntries RPC containing the votes it got during the election, and if the latter, it will decrement the nextIndex for that node and retry, just as in Raft.

D. Leader election

Like Raft, BFT Raft uses randomized timeouts to trigger leader elections. The leader of each term periodically sends heartbeat messages (empty AppendEntries RPCs) to maintain its authority. If a follower receives no communication from a leader over a randomly chosen period of time, the *election timeout*, then it becomes a candidate and initiates a new election.

In addition to the spontaneous follower-triggered elections, BFT Raft also allows client intervention: when a client observes no progress with a leader for a period of time called the *progress timeout*, it broadcasts UpdateLeader RPCs to all nodes, telling them to ignore future heartbeats from what the client believes to be the current leader in the current term. These followers will ignore heartbeat messages in the current term and time out as though the current leader had failed, starting a new election. The UpdateLeader RPC includes the client's signature and the current leader id. When a node receives an UpdateLeader RPC with a valid client signature, it ignores the future heartbeats from the leader of the current term only if the leaderId matches. Otherwise, it rejects the request and replies with the current leader.

To begin an election, a follower increments its current term and sends RequestVote RPCs in parallel to each of the other nodes in the cluster asking for their vote. RequestVote RPCs themselves work similarly to Raft. The modifications come primarily in the recipient of a RequestVote RPC.

When a node receives a RequestVote RPC with a valid signature, it grants a vote only if all five conditions are true: (a) the node has not handled a heartbeat from its current leader within its own timeout (b) the new term is between its current term + 1 and current term + H, (c) the request sender is an eligible candidate, (d) the node has not voted for another leader for the proposed term, and (e) the candidate shares a log prefix with the node that contains all committed entries.

A node always rejects the request if it is still receiving heartbeat messages from the current leader, and it ignores the RequestVote RPC if the proposed term has already begun or if it is larger than current term plus the high-term water mark H. We choose H such that it is statistically unlikely to encounter H consecutive split votes when a leader has actually failed (i.e., among nodes that only start elections when necessary). Spurious elections are ignored because votes are not granted unless a node has noticed that the leader is unresponsive.

If a RequestVote is valid and for a new term, and the candidate has a sufficiently up to date log, but the recipient is still receiving heartbeats from the current leader, it will record its vote locally, and then send a vote response if the node itself undergoes an election timeout or hears from a client that the current leader is unresponsive. We call this

technique *lazy voting*, because nodes wait until they believe an election needs to occur before ever casting a vote. Once a vote is sent, the node will update its term number. It does not assume that the node it voted for won the election however, and it will still reject AppendEntries RPCs from the candidate if none of them contain a set of votes proving the candidate won the election.

In a cluster with $f > 1$ a candidate needs to keep track of multiple nodes that the client has told it may be faulty, in order to prevent these nodes from alternating leadership without a non-failed node ever becoming leader. For $f = 1$, just ignoring the current leader's RPCs is sufficient, and this extra state can be discarded once a new leader is elected.

The candidate continues in the candidate state until one of the three things happens: (a) it wins the election, (b) another node establishes itself as a leader, or (c) a period of time goes by with no winner (i.e., it experiences another election timeout).

A candidate wins an election if it receives votes from a quorum of the nodes. The candidate then promotes itself to the leader state and sends heartbeat messages with the votes and the updated term number to establish its authority and prevent new elections. The signed votes effectively prevents a byzantine node from arbitrarily promoting itself as the leader of a higher term. Followers that receive this heartbeat message will update their leaderId and term if the leader presented enough signed votes for the matching term.

While waiting for votes, a candidate may receive an AppendEntries RPC from another node claiming to be leader. If the leader's term is at least as large as the candidate's current term and the leader provides enough votes to support its authority, then the candidate returns to follower state.

E. Correctness Arguments

Safety - Message signatures: All RPCs from clients and replicas are signed by a private key known only to that client or replica. Every node in the system can verify these RPCs with the node's public key. Further, client public keys are separated from replica public keys, so replicas cannot produce valid client commands.

Liveness - Client intervention: If a Byzantine leader ignores all inbound requests, the cluster becomes unavailable to clients. BFTRaft uses client intervention to restore availability in this situation: a client indirectly initiates a new election if no progress can be made. With $f = 1$ this is sufficient to cause a non-failed node to become leader after a Byzantine-failed node was leader. With $f > 1$, replicas must maintain additional state to ensure that Byzantine nodes do not alternate leadership and compromise liveness.

Safety - Incremental hashing: A Byzantine leader can arbitrarily forge, modify, delete, duplicate, or reorder client requests in the Raft algorithm. BFT Raft uses cryptographic signatures to prevent Byzantine leaders from forging or modifying client requests, but a Byzantine node could still reorder valid requests, and cause one ordering to be replicated on one set of nodes, and another ordering on a different set. With incremental hashing, nodes can be certain that both the contents and ordering of other node's logs match their own, and commit log entries safely with an agreed-upon ordering.

Liveness - Election verification: A Byzantine node can starve the whole cluster by claiming to be a leader of a higher term number than the current leader. This is avoided by requiring new leaders to prove to other nodes that they won an election in their first AppendEntries RPC to each node in a new term.

Safety - Commit verification: A Byzantine node can decide to arbitrarily increase the commit index of other nodes before log entries have been sufficiently replicated, thus causing safety violations when nodes fail later on. BFT Raft shifts the commit responsibility away from the leader, and every node can verify for itself that a log entry has been safely replicated to a quorum of nodes and that this quorum agrees on an ordering.

Liveness - Lazy voters: Even if nodes cannot convince other nodes to follow them without an election, Raft is still susceptible to nodes starting new elections when they are not necessary. BFT Raft solves this problem by having nodes only cast votes when they would otherwise become a candidate. This way, a quorum of votes signifies both that a node has a sufficiently up to date log to lead the cluster, but also that a quorum of nodes believed that a new term was necessary to make progress.

III. Implementation

We have implemented a proof-of-concept of this algorithm in Haskell. The code is available under an open-source license from `https://github.com/chrisnc/tangaroa`. To begin the implementation we started with the basic Raft algorithm as described by Ongaro [1]. We aimed to make this design as modular as possible, and so we provide ways for other application programmers to supply their own transport and message serialization functions. This is done using a simple dependency injection via a Reader monad transformer. The core Raft implementation is agnostic to all details about how nodes can be contacted and named, what messages actually look like, and what the possible state machine commands and results are. Importantly, the transport and serialization functions can be swapped independently of the state machine commands and results.

For our proof-of-concept, we let nodes be identified by an IP address and UDP port number pair, and we serialized messages using the `Show` and `Read` typeclasses. These choices were made for simplicity and debuggability. Serializing to human-readable strings made commands easy to inspect with network utility programs like Wireshark and Netcat. In the BFT Raft implementation, we found it was necessary to have a more compact serialization format, so we also use Haskell's `DeriveGeneric` mechanism and the `binary` package to automatically generate serialization functions for the various RPCs in the implementation.

Most of the core Raft logic is implemented in a custom monad that consists of a `Reader` and `State` monad transformer on top of the `IO` monad. Working in the `IO` monad is necessary in many of Raft's core behaviors, which

involve sending network packets and writing persistent state to disk. In a few places we do call out to pure functions where convenient. The `Reader` transformer is used to store things like cluster configuration, public keys of other replicas and clients, as well as the node's private key. The `Reader` transformer is also where all of the swappable functions for serializing and sending messages, reading/writing persistent state, applying commands to the state machine, etc. are accessed. This allows the core Raft logic to refer to these abstractly.

The `State` transformer is used to maintain the volatile state of the Raft algorithm (as well as an in-memory copy of the persistent state). This includes things like the current role, term, commit index, etc. Candidate- and leader-specific state is also stored here. The `State` transformer also contains a reference to a timer thread, which can be started, stopped, and reset as needed to trigger election or heartbeat events.

The implementation makes extensive use of the `lens` package to provide concise manipulation of and access to nested structures in the Raft state and configuration. The implementation uses a handful of compiler extensions as well; the most important of these being: `DeriveGeneric`, `GeneralizedNewtypeDeriving`, and `RecordWildCards`.

We described the usefulness of `DeriveGeneric` for automatic serialization with the `binary` above. If other applications want it, they can use other packages like `aeson`, `cereal`, which also use `DeriveGeneric` to provide automatic serialization to other formats.

Because Raft uses a few different kinds of numbers with different, mutually-exclusive uses, (log indices, term numbers, request IDs), we use `newtype` wrappers around the latter two to cause code that uses values in an invalid context fail to compile. The `GeneralizedNewtypeDeriving` extension lets us do this trivially while still letting us treat these values as ordinary numeric types with no additional runtime cost. Using `newtype` wrappers around the different kinds of numerical identifiers prevented at least one bug at compile time in the course of implementing our proof-of-concept; a log index and a term number were swapped in the code for checking whether a node should cast a vote.

The `RecordWildCards` extension is used throughout our code for easily deconstructing RPCs, which are defined as Haskell records with several fields each. This allows the code to be written as though each field of the RPC were a standalone input argument, even though a function only takes one argument for the whole record. This improved the conciseness and readability of the implementation.

With our state machine-agnostic library, we implemented a simple key-value store in under 100 source lines of code, as measured by `sloccount`. One module defines the type of commands available to run, (insert, get, set, delete), and the possible results, (a value, success, failure). Another module implements a simple client that gets commands from standard input and prints results to standard output. A third module implements a simple server that maintains a map from key strings to value strings and updates the map appropriately for each command type, producing a value for a successful get, a success for a valid insert, set, or delete, and a failure otherwise. This code is completely oblivious to the details of UDP, string/binary serialization, and the Raft algorithm itself (including whether it uses the BFT version or not).

IV. Improvements and Weaknesses

With more time, our implementation could be made more flexible and robust. The current implementation does not bound the size of messages, which is a problem for UDP, the only transport protocol we used. We also would have liked to generalize the BFT implementation to properly implement handling successive client requests to ignore nodes. The current implementation can only be considered to guarantee liveness for one Byzantine failure, as two Byzantine nodes could collude and alternate leadership while preventing clients' progress.

We also think that borrowing more heavily from the techniques in PBFT would have improved the simplicity and robustness of our design; particularly, simplifying leadership succession using round-robin by term number would remove a significant amount of mechanism from our leader election process, which is very vulnerable to Byzantine behavior without significant modifications. With round-robin leadership, the possibility of split votes is avoided completely, and we can sidestep the problem of the Byzantine nodes alternating leadership without a non-failed node in between.

Lastly, implementing a more complex application on top of our library would have been a useful exercise to test how viable the total separation of the Raft algorithm from state machine logic is in practice.

V. Conclusion

We have proposed a Byzantine fault tolerant extension to the Raft consensus algorithm, with various techniques for guarding against threats to liveness and safety that can arise in the presence of Byzantine faults. These included cryptographic mechanisms for proving the consistency of logs and of vote quorums. Threats to liveness are more complex to handle, and we need to introduce a series of modifications that ensure that as long as no more than f nodes are behaving arbitrarily, then the remaining nodes can make progress. This included giving clients the power to force an election, requiring new leaders to prove an election victory, and delaying votes until nodes believe an election is necessary. These modifications come at the cost of additional complexity and overhead, but the core architecture of Raft is mostly intact.

References

[1] Ongaro, Diego, and John Ousterhout. "In Search of an Understandable Consensus Algorithm." Draft on October 7 (2013).

[2] Castro, Miguel, and Barbara Liskov. "Practical Byzantine Fault Tolerance." OSDI. Vol. 99. 1999.