Battleship: Byzantine Fault Tolerant Raft

Mitchell Dumovic and Saachi Jain

***Abstract*— We propose Battleship: a Byzantine Fault Tolerant version of the consensus protocol Raft [1]. Battleship retains Raft's centralized nature, aligning with Raft's original objective of an understandable consensus algorithm while maintaining safety, fault tolerance, and liveness under weak synchrony in a Byzantine environment. We implement a prototype of Battleship in Python, building off a previously existing open source implementation of Raft called Zatt [3].**

I. Introduction

Raft is a consensus algorithm used to manage a log replicated onto multiple machines. Raft was designed as a highly modularized protocol that would be easier to understand than Paxos. In particular, Raft is highly centralized, with a leader that can overwrite log entries on followers in order to maintain consistency. Raft is safe, fault tolerant, and live (under timing assumptions).

In systems with Byzantine behavior, a number of faulty nodes can behave in an actively malicious manner. Such an environment could occur due to cyber attacks, errors, or consensus involving untrusted peers. In this paper, we discuss Battleship: a byzantine fault tolerant adaptation of Raft. We first discuss how Battleship implements both log replication and leader election. We then discuss our implementation of Battleship in Python.

A. Related Work

There is a significant body of work dedicated to creating Byzantine fault tolerant consensus systems. Of particular significance is the Practical Byzantine Fault Tolerance [2] (PBFT) protocol, which extends Paxos to handle malicious nodes. PBFT is highly decentralized to mitigate the damage of a faulty primary; other than receiving the initial request, the primary plays no role in coordinating consensus. Instead, PBFT uses broadcasts to ensure safety in the face of faulty nodes.

However, broadcasts congest the network and lead to high latency. Battleship seeks to avoid this issue by maintaining the highly centralized nature of Raft. Battleship has all operations funnel through the primary via a three phase protocol. Moreover, unlike PBFT, Battleship avoids situations where every node must broadcast to every other node, resulting in a costly n^2 number of messages in the network.

II. Providing Fault Tolerance in Raft

A. Raft Overview

Raft provides a centralized consensus protocol to create a replicated state machine. The state machine is implemented using a replicated log, where the log contains a sequence of commands from a client. Raft guarantees that these properties are true at any time:

1) **Election Safety**: at any time, there can only be one leader for a particular term
2) **Leader Append-Only**: a leader only appends to its log, never overwriting or deleting
3) **Log Matching**: if two nodes contain an entry at the same index with the same term, then the logs are identical up to that index.
4) **Leader Completeness**: if a log entry is committed, then that entry must be present in the log of any leader thereafter
5) **State Machine Safety**: If a server applied a log entry to its state machine at an index, no other server should apply a different entry at that index.

Each of these properties are extended in Battleship to only include non-faulty nodes, as we have no control over a faulty node's behavior.

B. Opportunities for Malicious Attacks in Raft

Malicious actors can break several of Raft's safety guarantees on both the log replication and leader election front. A faulty leader could prevent safety or liveness by ignoring client calls or falsifying responses. Within the consensus protocol, a faulty node could falsify a quorum by responding twice to two conflicting requests, allowing conflicting entries to be committed on non-faulty nodes. A faulty leader could delete committed entries by instructing backups to overwrite them, or instruct replicas to commit entries that have not been replicated.

In leader election, a faulty actor can vote for two leaders and violate the election safety property. Faulty nodes could also ensure that the leader is always faulty

by delaying the votes of any other candidate, preventing the system from making progress.

C. Battleship Overview

We thus aim to create a protocol that remains safe in the face of f faulty nodes. Faulty nodes can behave arbitrarily and send incorrect messages. We also assume that the adversary has power over the network: the adversary can arbitrarily delay messages from any node (including messages from non-faulty nodes). However, we assume weak synchrony, in that messages cannot be delayed indefinitely. More precisely, we assume that the delay in a message cannot grow faster than a polynomial function of time [2]. We introduce exponential back-off in our timeouts for each term so that eventually a term will exist where a non-faulty leader must be elected under this timing assumption.

We assume that the client is trusted, and that any genuine request that the client makes can be applied to the log. Although outside the scope of this paper, Battleship could be extended to perform access control as long as each replica could identify the permissions associated with a client.

D. A Non-Faulty Quorum

To provide byzantine fault tolerance, there must exist a quorum Q such that the system remains both live and safe after f failures. The minimum number of replicas needed to ensure byzantine fault tolerance in an asynchronous system is $N = 3f+1$, with a quorum size $Q = 2f+1$ [2]. This ensures that a quorum exists even if f nodes fail, and that the intersection of two quorums includes at least one non-faulty node. Thus, we assume there are $3f + 1$ replicas for a Battleship deployment designed to withstand f byzantine failures.

E. Cryptography

Battleship relies on quorums to provide safety guarantees. However, quorums only work if messages cannot be spoofed; otherwise, a faulty node could simply manufacture messages to artificially satisfy a quorum. Thus, we use RSA digital signatures to sign all messages. The client and each of the nodes have a private and public key. The public keys are known by all members. One could use a trusted key server to retrieve these public keys; the exact method to distribute these public keys is beyond the scope of this paper.

Define $D(m)$ as the SHA-256 hash of the message m. Rather than signing an entire message, members compute the hash of the message and sign the hash. Thus, we denote $\langle m \rangle_i$ as a message signed in this manner by node i. $\langle m \rangle_L$ is a message signed by the current leader, and $\langle m \rangle_C$ is a message signed by the client. Whenever a member q receives a message $\langle m \rangle_p$ from node p, q uses p's public key to verify that the message was actually send by p. If the verification fails, then q ignores that message. Thus, faulty nodes cannot spoof messages from other nodes.

III. Log Replication

One of Battleship's main challenges is implementing log replication in the presence of faulty servers.

A. Making a request

In order to initiate an operation, the client creates a request $\langle r \rangle_C$ and sends it to what it believes is the current leader. After verifying the client's signature, the node i receiving the request checks to see if the request has already been committed. If so, node i replies directly back to the client with proof that the operation has been committed. If node i is not the current leader, i replies back to the client with a $\langle \texttt{REDIRECT}, addr_l \rangle_i$, where $addr_l$ is the address of the current leader. Thus, the client can contact the leader itself.

If the client does not hear the outcome of its requested operation after a certain time period, the client broadcasts the request to all nodes. This serves two functionalities. First, the broadcast ensures that the request will be sent to the current leader, even if a node lies in a `REDIRECT` message. Secondly, this broadcast triggers the time-out for leader election in the case of a faulty primary (See section IV).

B. Defining an Entry

Like in Raft, an entry is defined by:

1) `data`: the data of the request, signed by the client
2) `term`: the term in which the request was issued

A single entry e can be in one of three phases:

1) **Pre-Prepared**: A pre-prepared entry is an entry that appears in the log.
2) **Prepared**: A prepared entry is an entry that, at one point, was pre-prepared on at least $2f + 1$ nodes. A prepared entry can only be overwritten by a `PREPARE` message (more below).
3) **Committed**: A committed entry has been stably replicated (prepared) on 2f+1 replicas. Entries which are committed can be applied to the state machine and externalized to the client.

Log replication occurs in three phases: `PRE-PREPARE`, `PREPARE`, `COMMIT`.

With these definitions, we redefine the Log Matching safety property of Raft as: if two nodes contain an entry at the same index with the same term that is *prepared*, then the logs are identical up to that index.

Each node keeps track of `currentTerm`, which is the last term it received a `TERM_CHANGE` message for (see below). Just as in the original Raft protocol, the leader keeps track of a `NextIndex` for each of its peers. The `NextIndex` is originally set to the last entry of the leader's log, and is backtracked in order to overwrite a follower's log.

C. Replicating a Value

When a leader receives a valid request $\langle r \rangle_C$ from a client, it validates the signature and appends the request to its log. This entry e is now pre-prepared.

Just as in the original Raft protocol, the leader keeps track of a `NextIndex` for each of its peers. The `NextIndex` is originally set to the last entry of the leader's log, and is backtracked in order to overwrite a follower's log.

After pre-preparing an entry e by appending it to its own log, the leader broadcasts $\langle$ `PRE-PREPARE`, `term`, e, `prevLogIndex`, `prevLogTerm` $\rangle_L$ to its followers. A follower will accept a `PRE-PREPARE` if all of the following are true:

1) e has a valid client signature
2) `term` is equal to `currentTerm`
3) There is an entry in the follower's log with index `prevLogIndex` and term `prevLogTerm`.
4) If e is overwriting another entry then that entry must not be prepared or committed.

Note that according to requirement 4, a leader cannot overwrite a prepared entry with a `PRE-PREPARE`. If follower i accepts the `PRE-PREPARE`, it deletes the existing entry at that index (and all that follow it), and appends e to its log. It then sends back $\langle$`ACK_PRE-PREPARE`$, e\rangle_i$ to the leader.

If the `PRE-PREPARE` fails because condition 3 does not hold, then this backup is out of date. The backup notifies the leader, and the leader decrements the follower's `NextIndex` and tries to `PRE-PREPARE` the entry previous to the failed `PRE-PREPARE`. This backtracking mechanism occurs exactly as it does in the original Raft protocol.

The leader collects $2f+1$ $\langle$`ACK_PRE-PREPARE`$, e\rangle_i$ from its followers (including itself). These messages are assembled into the object P_e, and provides proof that $2f + 1$ peers in the network have pre-prepared e. The leader then marks the entry on its own log as prepared and broadcasts $\langle$`PREPARE`$, e, P_e\rangle_L$ to its followers.

A follower i accepts a $\langle$`PREPARE`$, e, P_e\rangle_L$ if the proof P_e contains at least $2f + 1$ valid `ACK_PRE-PREPARE` messages for that entry from different peers. It then overwrites the existing entry at that index with e (if e was not already there) and marks e as prepared. It then sends back $\langle$`ACK_PREPARE`$, e\rangle_i$.

The leader collects $2f + 1$ $\langle$`ACK_PREPARE`$, e\rangle_i$ messages. and messages assembles them together into C_e which serves as proof that $2f + 1$ peers in the network have prepared e. Now the entry has been stably prepared, and can thus be committed and externalized. The leader marks the entry as committed, applies it to its state machine, and sends back $\langle$`RESULT`$, C_e\rangle_L$ to the client. Notification of commit, along with the proof C_e, is piggy-backed to the followers in the next `PRE-PREPARE`, at which point the followers commit the entry, apply e to their state machines, and store C_e for future reference.

D. Log Replication in Practice

While a three-phase approach makes sense for explanatory purposes, in practice these phases can be bundled into a single `UPDATE` message as follows.

Instead of keeping the replication state (pre-prepared, prepared, and committed) per entry, each node keeps track of three indices:

1) `CommitIndex`: index of the last commit
2) `PrepareIndex`: the index of the last prepare
3) `PrePrepareIndex`: the index of the last pre-prepare

Naturally `CommitIndex` $\leq$ `PrepareIndex` $\leq$ `PrePrepareIndex`. Entries up to and including `CommitIndex` have all been committed. Entries in the interval (`CommitIndex`, `PrepareIndex`] have been prepared. Entries after `PrepareIndex` have only been pre-prepared.

An `UPDATE` message to a peer contains the log entries from the peer's `NextIndex` up until the end of the leader's log, the leader's `PrepareIndex` and `CommitIndex`, and a proof P. P contains the latest responses from the peers to the leader's `UPDATE` and prove that both the `PrepareIndex` and the `CommitIndex` are valid. A follower then performs the three phases of verification from the above section, updating its log and moving its `PrepareIndex` and `CommitIndex` appropriately.

A precise description of the `UPDATE` is described in RPC 1. Bundling the 3 phases into one update

reduces the size and number of messages sent across the network. Moreover, with a consolidated update, it is possible to pre-prepare, prepare, and commit multiple entries at once. This improves performance as each entry does not need to work through the sequence of pre-prepare, prepare, and commit sequentially.

RPC 1 UPDATE

Invoked by leader to replicate onto followers

Arguments:

- **entries[]**: log entries from nextIndex onward
- **prevLogIndex**: nextIndex - 1
- **term**: leader term
- **leaderPrepare**: the leader's prepare index
- **leaderCommit**: the leader's commit index
- **proof**: Map of peer $\rightarrow$ peer's latest `UPDATE` response

Response:

- **prePrepareIndex**: this peer's updated prePrepareIndex
- **logHash**: SHA-256 digest of the log up through the prePrepareIndex
- **prepareIndex**: this peer's updated prepareIndex
- **logPrepareHash**: SHA-256 digest of the log up through the prepareIndex

Receiver implementation

- Validate the signatures on the entries and the proof
- Create a "hypothetical log" which is what the peer's log would look like with the entries added
- For each message in the proof, re-calculate the logHash and the logPrepareHash on the relevant entries of the hypothetical log. If the logHash or logPrepareHash does not match, ignore that message of the proof.
- Validate that there are $2f + 1$ valid messages in the proof with a prePrepareIndex $\geq$ leaderPrepare. Also validate that there are $2f + 1$ valid messages with a prepareIndex $\geq$ leaderCommit.
- Check to make sure that leaderPrepare $\geq$ the peer's prepareIndex. If not, the leader is trying to overwrite a prepared entry with a pre-prepare, so the receiver should reject.
- Replace the peer's log with the hypothetical log. Prepare up to the leaderPrepare and commit up to the leaderCommit. Store the proof sent as proof of commit.

IV. Leader Change

A. Foregoing Leader Election in Favor of Round Robin

Naive leader election is problematic in a system with malicious nodes. In any leader election scheme, faulty nodes can choose to delay messages of all other candidates except itself and always become leader. Other nodes have no way of knowing that they need to choose someone other than a faulty node. Therefore, the faulty node can just make sure its message is sent out first and therefore will always be elected.

We thus forego Raft's leader election in favor of a round robin algorithm influenced by PBFT's `VIEW-CHANGE` process. We assign each node an identification number. Let n be the total number of nodes (i.e $3f + 1$). For each proposed term t, only the node with identifier $t \mod n$ can be the leader. Since we cycle through leaders, faulty nodes cannot dominate the leader election.

B. Transitioning to a New Term

If the client has not heard a successful response to its request from the primary, the client broadcasts the request to all backups. Unless the request has already been committed, the backups start a timer to trigger a term change. We use timers with exponential backoff; after a timer expires, the timer is reinstated with twice the timeout until progress is made. The timer is canceled and the timeout is reset if the backup receives a successful command to commit an entry.

When a backup i times out, it calculates a $\texttt{proposedTerm} = \texttt{currentTerm} + 1$. It then sends to the proposed leader of `proposedTerm` the message $\langle$`TERM_CHANGE`, `commitIndex`, P, `entries` $\rangle_i$ where `commitIndex` is the backup's commit point, P is proof of commit and prepare, and `entries` is are the prepared entries that have not been committed.

The new proposed leader L' gathers $2f$ `TERM_CHANGE` messages from its peers. L' validates `commitIndex` and attached prepared `entries` using P, and then compiles these messages into P_{TC}, which provides proof that L' is a validly elected leader. Just as in Raft, the leader must have the most up to date log. However, from the `TERM_CHANGE` messages, L' has the information it needs to construct the most up to date log even if the node was not the most up to date before the term change. For each `TERM_CHANGE` message m:

1) Starting at `m.commitIndex` with the attached `m.entries`, L' creates a hypothetical log. It

then re-computes the log hashes on this hypothetical log. If L' discovers that it does not have the entry corresponding to the latest committed value, it invalidates itself as leader and we must wait until the next term to get work done.
2) Otherwise, L' adds the hypothetical log as a candidate log.

L' picks the candidate log that is most up to date, as defined in the Raft paper. In particular, it picks the log with, in order of priority:

1) The highest commit index
2) The last prepared entry with the highest term number
3) The highest prepared index

L' then broadcasts the message $\langle$ `NEW_TERM`, `proposedTerm`, $P_{TC}\rangle_{L'}$ becomes the new leader. Backups who hear a valid `NEW_TERM` proposing a term higher than their current term accept the new leader, adjusting their currentTerm to the proposed term.

If the backup times out again because no progress has been made, the backup increments its proposedTerm and sends out another `TERM_CHANGE` to try to elect the next leader in the round robin.

V. Correctness

In order to prove that Battleship is correct, we show that our protocol both maintains Raft's safety properties and allows the system to make progress under weak synchrony.

A. Election Safety:

Battleship round robins through potential leaders via the term change protocol. Thus, there can only ever be one leader assigned to a given term. Moreover, since the majority of nodes must agree on a term and the term number is chosen sequentially, a faulty node cannot simply choose a term number that makes it the leader.

B. Leader Append Only

Our protocol continues to have leaders only append to their log. A faulty leader could decide to overwrite a value, but will not be able to unilaterally override a prepared value and thus cannot write over entries that could be externalized.

C. Log Matching

The log matching property is satisfied if, when two non-faulty logs share an entry that is prepared with the same index and term, the logs are identical up until that term. This has to occur because entries are prepared in relation to a hash of the log up until that entry. More specifically, a `PREPARE` command contains $2f+1$ pre-Prepare commands whose log hashes all have to match. Thus, if any node prepared an index due to a `PREPARE`, they must have an identical log to the prepare proof up until that index.

D. Leader Completeness and State Machine Safety

We first prove the following property: if an entry is committed by any non-faulty node, $2f$ other nodes must have that node prepared or committed and (for at least $f+1$ of those nodes) that entry will never be overwritten. This is because a committed index must have been prepared by $2f+1$ nodes. A prepare can only be overwritten by a `PREPARE` message. In order to overwrite a prepare, a leader must have evidence that $2f+1$ nodes are willing to `PRE-PREPARE` over the entry. However, it is impossible for any leader to garner those $2f+1$ `PRE-PREPARE` votes since $2f+1$ out of the $3f+1$ have promised not to `PRE-PREPARE` over the entry, and of those only f can be faulty.

Given this property, we can now prove both Leader Completeness and State Machine Safety. Suppose that a leader has committed an entry e. That means there is a set of $2f+1$ replicas who have either stably committed or prepared e (as above). A new leader needs $2f+1$ `TERM-CHANGE` messages to start a new term. Since two quorums must intersect at a non-faulty node, there must be a non-faulty node (let's call it i) who stably committed or prepared e and participated in the term change. Since that entry cannot be overwritten, there cannot exist a node with a more up to date log than i that does not contain e. Therefore the leader (if it is non-faulty) will construct a log with e. Even if the new leader is faulty, since e cannot be overwritten, e will continue to exist on $2f+1$ replicas so the same properties will continue to hold until a leader election commences with a non-faulty leader. Thus, leader completeness holds.

State machine safety is a direct result of the property we proved above. Entries are only applied by non-faulty nodes when they are committed. Since a committed entry cannot be overwritten or prepared with a later value, if an entry was applied to the state machine of a non-faulty node, no other node will apply a different entry at that index.

There are a few miscellaneous safety properties that need to be added in the byzantine environment. Firstly, a node cannot lie to the client about a commit, because that node needs provide proof of commit. Secondly,

since we use cryptographic signatures, a leader cannot forge a client request. As stated above, we assume that the client is trustworthy; Battleship loses its safety properties if a trusted client can send malicious requests.

E. Liveness

Under weak synchrony, Battleship always has the potential to terminate. This is because leader elections are triggered when no progress has been made. Since leader elections are conducted with exponential backoff, there will eventually exist a proposed term for a non-faulty leader where the adversary cannot delay the messages to prevent a term change. Once a non-faulty leader is elected, it can make progress toward committing the request.

Moreover, a non-faulty leader will never become stuck when trying to commit a new request. A leader only fails to replicate a request if $f + 1$ nodes have already prepared another value at that index, preventing the leader from getting $2f + 1$ `ACK_PRE-PREPARE`. However, if $f + 1$ nodes prepared a value, that value would have appeared in the term change, so the leader would already have that value and would not try to overwrite it.

VI. Implementation Details

We implemented a prototype of this algorithm in Python. The full source code of our implementation is available at https://github.com/scoutsaachi/zatt. Our prototype is built off of an existing open source implementation of non-BFT Raft called Zatt[1]. Zatt includes log replication, leader election, membership changes, and log compaction. For our prototype, we did not implement membership changes and log compaction. Our prototype uses the asyncio module to periodically send messages and set timeouts. We used Python's pycrypto module to generate hashes and public/private keys.

A. Server Nodes

On startup, each server node loads a configuration file containing the the (IP, Port) pairs for the cluster and the location of a directory containing that node's private key, the public keys of every other node, and the public keys of any trusted clients.

Each server can be in one of three states: `Leader`, `Follower`, and `Voter` extending a generic `State` class. The Leader and Follower behave as described in the log replication sections above. A Follower switches to the Voter class on time-out. Voters ignore all update messages sent from leaders in any term and send periodic `TERM_CHANGE` messages to the leader for the next proposed term. Each server has an `Orchestrator` responsible for transitioning the node between states and sending and receiving messages to and from clients and peers.

B. Client Nodes

The client is implemented through a simple key-value store class called a `DistributedDict`, which subclasses a normal python user dictionary. The `DistributedDict` is initialized with the cluster information, the public keys for each of the nodes in the cluster, and the client's secret key used to sign messages to the leader. When an entry is added to or deleted from the dictionary, the client sends a request to the node it believes to be the leader to add an entry to the state machine for that request. If the client receives a redirect response, it updates the value of what it believes to be the current leader accordingly. If the client does not receive a valid response after a certain time, it broadcasts the request to all nodes in the cluster.

VII. Conclusion

We presented *Battleship*, a consensus protocol based off Raft which implements consensus in a system where nodes can exhibit Byzantine failure. Battleship maintains the safety, fault tolerance, and liveness properties of Raft by adding an extra phase to the commit protocol. Additionally, Battleship forces all nodes and clients in the system to cryptographically sign messages, and only updates commit and prepare points when given proof. Timeouts and elections are set such that a faulty node cannot permanently prevent progress in the system. Overall, Battleship maintains the Raft's centralized nature while extending it to handle malicious actors.

References

[1] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC 14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305319, 2014

[2] M. Castro, B. Liskov, Practical Byzantine Fault Tolerance, *3rd OSDI*, 1999.

[3] Simon Accascina, *Zatt*, (2017), GitHub repository, https://github.com/simonacca/zatt