BFT Raft

Danny McCormick and Venkat Nagarajan

Raft Review

- Simple/easy to understand consensus algorithm
- Assumes failure only by stopping
- Vulnerable to Byzantine nodes

Raft Mechanics

- Main components are leader election and log replication
- Leader election guarantees election of *exactly* one leader
- Log replication guarantees logs can’t be overwritten, majority of nodes have replicated committed logs, logs are identical if they share same commit index, and leader will have all logs committed during its term

Basic Control Flow

- 3 main components
  - Prop_node.py
  - Network_ctrl.py
  - Client.py
- Prop_nodes run BFT Raft
- Client sends requests over network to current leader
- Leader replicates request across network (normally some processing would happen here too), and returns result.

BFT Raft

- Built on top of Raft
- Adds several key features to protect against Byzantine nodes
  - Cryptographic message signatures and Incremental Cryptographic Hashing
  - Client Intervention
  - Election Verification
  - Commit Verification
  - Lazy Voters

Cryptographic Message Signatures/Hashing

- Message Signatures used to ensure byzantine nodes can’t spoof messages from other nodes
- Incremental hashing used to ensure that when a node replicates the log, it has replicated the entirety
- Due to lack of experience with cryptography, we skipped this
- We operated under the assumption that messages weren’t spoofed and node’s won’t partially replicate the log

Client Intervention

- If a Byzantine node gains leadership, it could stall system by not relaying Client requests
- Addressed by adding a timeout - if client doesn’t hear back from leader within timeout it triggers an election

Election Verification

- During an election, a Byzantine node could claim leadership even if it didn't receive the majority of votes
- Fixed by adding an election verification stage
  - Leader must send out all nodes that voted for it. Other nodes then verify with each other that leader actually received those votes

Commit Verification

- Byzantine leader could break consensus by sending different messages to be appended to different nodes
- To fix this, when followers receive append messages, they send their acknowledgement to the other followers along with the original append message
- Only when they receive matching messages from a majority of nodes do they actually commit the changes

Lazy Voters

- A byzantine node could stall out the system by repeatedly triggering elections
- To fix this, nodes only vote in a new election if they think the leader is faulty
    - Controlled by using a watchdog that starts new election on a timeout

Testing Methods

- Used a mininet network
- Created adversarial nodes that exhibited various Byzantine behaviors to run against

Example demo/test

- Uses an adversarial node that tries to claim leadership with a fake quorum
- Should be rejected because it didn’t actually win that quorum

Example demo/test

- Uses an adversarial node that tries to repeatedly trigger elections
- Should be ignored because leader is not faulty

Challenges Faced

- Significant slowdown
  - BFT Raft requires sending way more messages and receiving more replies than normal Raft
- Contention when declaring candidacy
  - Originally with the control flow, making voters lazy led to all nodes repeatedly declaring candidacy at once
  - Fixed by only allowing a node to declare candidacy once every 3 seconds

Lessons Learned

- The tradeoff of speed vs Byzantine Fault Tolerance is significant
- The tradeoff of simplicity vs Byzantine Fault Tolerance is also significant
- If you’re ok with the system being blocked, BFT gets easier
  - Half the features implemented were solely focused on making sure the system makes progress, other half were on maintaining correctness
  - In some cases might be possible to manually fix these issues by removing the faulty nodes, reducing some of the necessary tradeoffs