Danny McCormick and Venkat Nagarajan

Repo : https://github.com/venkatvandy/Distributed/tree/master/PBFTRaft

BFT-Raft Design

Overview

We will implement the Byzantine-fault-tolerant algorithm as described in *Tangaroa: A Byzantine Fault Tolerant Raft* by Christopher Copeland and Hongxia Zhong. We will start with an existing Python Raft implementation and extend it to be Byzantine Fault Tolerant. While it uses a very similar system, BFT Raft adds the following things: cryptographic message signatures, client intervention, incremental cryptographic hashing, election verification, commit verification, and lazy voters. Essentially, we will be adding these key features to take the algorithm from being non-Byzantine-fault-tolerant to being Byzantine-fault-tolerant assuming 3f+1 nodes. It appears that are currently no public BFT-Raft implementations in Python.

Existing Code

While there are many existing Raft implementations out there, most of the Python ones are either incomplete or buggy. Because of this, we will be building on top of a Raft implementation done for a prior class. This is a slightly stripped down version of Raft

Implementation breakdown:

We divided the implementation broadly into three files:

1. **Prop_node.py**
2. **Network_ctrl.py**
3. **Client.py**

Network.ctrl.py

It contains the network implementation i.e it has the methods defined to send messages across different
servers based on their IP address. The important methods are listed below:

- **wait_for_ctrl_connections** – This method is run in a background thread in each server so that servers can read messages from other servers.
- **send_ctrl_message_with_ACK** – This method has 5 parameters :
    - Message - The data you want to pass.
    - messageType – What type of message it is.
    - extra – Any extra data you want to send.
    - requestNode – The node to which you want to send this message.
    - timeout – the timeout value

Prop_node.py

This is the main file where the BFT Raft algorithm is implemented

Main methods:

- **join_network** – Used by servers to join a cluster
- **start_leader_election** – Used by a server to start leader election
- **stabilization_routine** – Used by servers to know about other servers in the cluster.
- **heartbeat_routine** - As soon as a server becomes a leader it starts sending out heartbeat signals to existing nodes in the system to show that it is alive.
- **leader_timeout_routine** - When a node in the system does not receive heartbeat signal from the leader within its timeout period, it announces itself as a CANDIDATE and starts election by calling start_leader_election

Client.py

Client has **two major tasks.**

- Send commands to be executed to the server.
- If the command takes too long to get executed it intervenes to start a new leader election.

New Functionality

In order to make it Byzantine Fault Tolerant, we will add cryptographic message signatures, client intervention, incremental cryptographic hashing, election verification, commit verification, and lazy voters. We will implement each of these in the following ways:

1) Cryptographic message signatures - Because we lack sufficient experience with cryptography, initially we will simply use the IP address of the sender as the message signature and not allow spoofing of messages. If time permits, we will extend this to a more robust and secure implementation.
2) Client intervention - The client must be able to trigger a new election if they have not heard back from the leader in a sufficient amount of time. To ensure this, we will have a client timeout for receiving replies. If it doesn't receive a reply within the timeout, it will broadcast a reelection mandate to all of the nodes.
3) Incremental cryptographic hashing - Again, due to insufficient knowledge of cryptography we will start with a simpler solution of simply mandating that all logs be copied. If time permits, however, we will extend the implementation to include this incremental cryptographic hashing which is used on top of each message to ensure that all logs have been appended.
4) Election verification - To ensure that a node can't claim to be a leader without winning the election, nodes will be mandated to send out a list of nodes that voted for them. All nodes will then verify that the candidate actually won before continuing by communicating with the other nodes that the candidate claimed won.
5) Commit verification - In order to ensure that entries are safely committed by all nodes, we will broadcast the append messages to all nodes and only commit when a quorum is reached. This takes the responsibility away from the leader.
6) Lazy voters - In order to ensure that a faulty node can't repeatedly trigger elections, nodes will only participate in elections if they think that they are necessary. This will be accomplished by only voting if the client intervenes or the node times out before receiving a message from the leader.

Testing

In order to test that our design is truly Byzantine Fault Tolerant, we will create code that can be run on a node to turn it into an adversary. This node will have several key capabilities in order to adequately test the system:

1) It will have the ability to be killed at any time to test that the consensus algorithm still works and can recover, regardless of whether the node is a leader or not.
2) As a leader, it will be able to be triggered to send out different values to different nodes to try to disrupt the consensus.
3) As a leader, it will be able to be triggered to block indefinitely to try to keep progress from being made.
4) It will be able to claim that it is the new leader whether or not it won the election to test the election process.
5) It will be able to repeatedly trigger elections to try to keep progress from being made.