CSE 4/586: Distributed Systems
Project Report: Phase 4

Safe Log Replication with leader election

Team Members:

Anindita Deb

Farhan Chowdhury

Subject Area: A Next Generation Communication Application

Introduction

In phase 1 we've implemented a simple user login application. The application tried to create a connection between client and server that both these client and server were bonded together within the same node. In phase 2 we have shown consensus between multiple nodes executing the same application with one node leading the others. In phase 3 we implemented Raft Consensus Algorithm to elect the leader, heartbeats and timeouts. In phase 4 we will be implementing safe log replication with leader election.

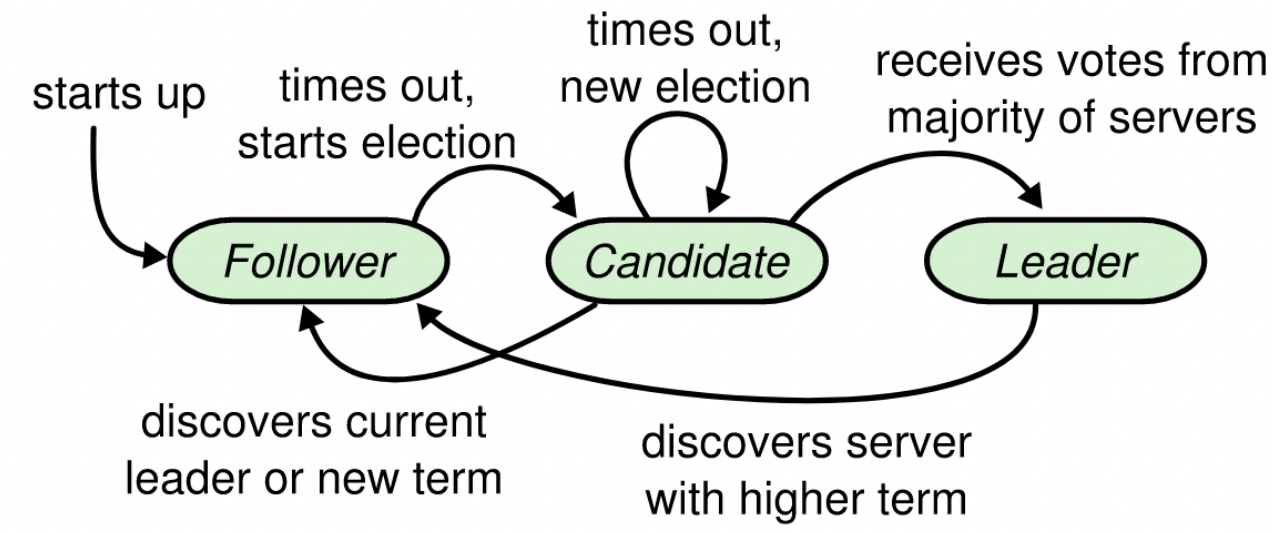


Figure: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

Motivation

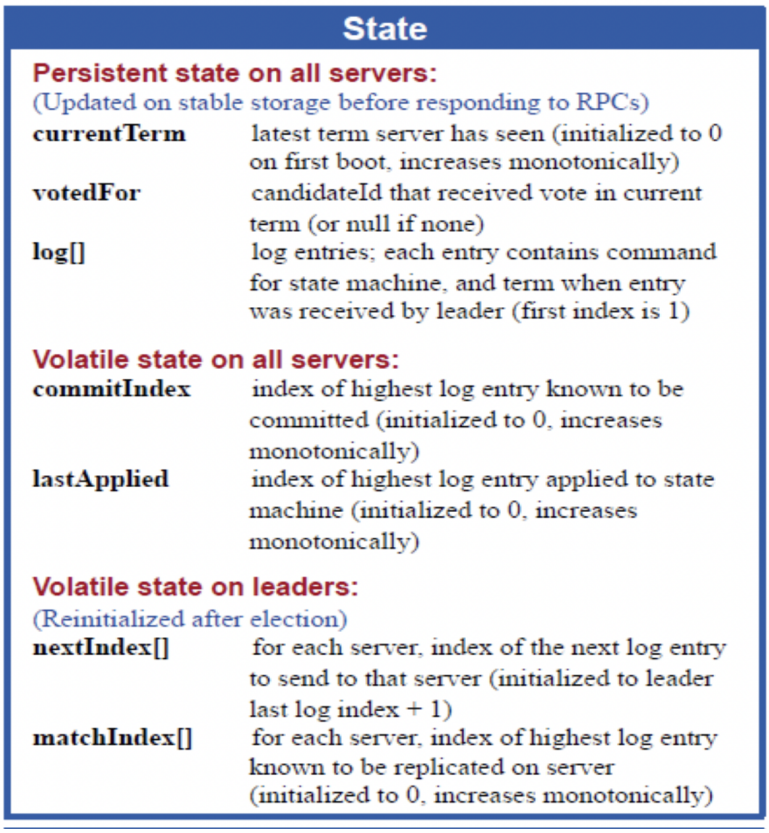
Phase 3 of the project was all about learning and implementing Raft consensus algorithm between different nodes. Multiple nodes were acting as servers and will hold 3 positions named leader, candidates and followers. The main motivation behind that phase was to see how a node gets elected as a leader and others become followers and candidates implementing Raft. Raft is basically a consensus algorithm. "Consensus is a fundamental problem in fault-tolerant distributed systems. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value, that decision is final. Typical consensus algorithms make progress when any majority of their servers are available; for example, a cluster of 5 servers can continue to operate even if 2 servers fail. We figured out how to elect the leader in RAFT. And for a successful leader election, we incorporated heartbeats,

timeouts, remote-procedure-calls (RPCs) on each node. We also implemented multiple asynchronous threads to introduce concurrency. We learned and implemented how to change the status of a node from a leader to follower, follower to candidate and back to being a follower.

In phase 3, "once a follower times out, it increases its own term number, converts to candidate state and sends out RequestVote RPC to all the other nodes. On the receiver side, the node when it receives a RequestVote RPC, it would compare its own term number with the candidate's term and it's own log entries with the lastLogIndex and lastLogTerm values (sent out during RequestVote RPC) and decide whether to give a vote or not. In phase 3, the lastLogIndex and lastLogTerm comparisons were trivial and not considered as the logs entries were always empty as no client requests were being made. However in this phase 4 we will see how these comparisons play an important role in deciding who can and who cannot become the leader" (Phase 3 project requirement).

Implementation

Each Servers(nodes) are storing the following information:



1) We have implemented 5 nodes here that would behave under the roles of a leader, followers, candidate using the RAFT algorithm in the backend.

2) Below is a snapshot of the code showing apppendEntryRPC & Heartbeats function. It shows the leader sends empty log entries also as heartbeats to all the other nodes to maintain its authority. In addition, it also replicates across all the nodes.

The leader needs to have dedicated asynchronous thread to send out AppendEntry RPC/heartbeats for which we have created two threads shown later.

```
def appendEntriesRPC(self, term, leaderId, prevLogIndex, prevLogTerm, entries, leaderCommit):
        print ('Heartbeat ' + leaderId + ' ' + str(self.currentTerm) + ' ' + str(term) + ' ' + str(entries))
        if term < self.currentTerm:
            return (self.currentTerm, False)
        elif term > self.currentTerm:
            self.initFollower(term)

        # Reset HearbeatTimeout
        self.electionTimeout = self.HearbeatTimeout()
        self.startTime = time.time()

        if prevLogIndex < len(self.log):
            entry = self.log[prevLogIndex]
            logTerm = entry["term"]

            if logTerm == prevLogTerm:
                self.log = self.log[:(prevLogIndex+1)] + entries

                if self.IndexUpdate < leaderCommit:
                    self.IndexUpdate = leaderCommit

                return (self.currentTerm, True)

        return (self.currentTerm, False)
```

3) Next we are trying to show two events -

- What happens when all the nodes/ servers are first started up, that is all the nodes begin as followers, since there is no leader.
- Secondly we are showing that when a leader fails to send the heartbeats to the followers it loses its authority and becomes a follower.

```
    # Make sure log entries on other servers are up to date
    def updateLeader(self):

        self.leaderLock.acquire()

        commitCount = 1
        for i in range(0, len(self.nodes)):

            n = self.nodes[i]
            prevLogIndex = self.nextIndex[i]-1
            prevLogTerm = self.log[prevLogIndex]["term"]
            entries = self.log[self.nextIndex[i]:]

            try:
                socket.setdefaulttimeout(self.socketTimeout)
                t,v = n.appendEntriesRPC(self.currentTerm, self.id,
                                         prevLogIndex, prevLogTerm,
                                         entries, self.IndexUpdate)
                socket.setdefaulttimeout(None)

                if t > self.currentTerm:
                    self.initFollower(t)
                    break
                if v == True:
                    self.nextIndex[i] = self.getLogIndex()+1
                    commitCount += 1
                elif v == False:
                    self.nextIndex[i] -= 1
```

```
def initFollower(self, term):
    self.state = 'Follower'
    self.setCurrentTerm(term)
    self.setVotedFor(None)
```

4) In the below code snapshot we are trying to show how the election starts once a follower's timeout occurs, the followers make transition to candidate state and start an election, sending RequestVoteRPC to other nodes.

```
def requestVoteRPC(self, term, candidateId, lastLogIndex, lastLogTerm):
        print ('Vote Request ' + candidateId + ' ' + str(self.currentTerm) + ' ' + str(term))
        if term < self.currentTerm:
            return (self.currentTerm, False)
        elif term > self.currentTerm:
            self.initFollower(term)

        if (self.votedFor is None) or (self.votedFor == candidateId):
            # Return False is voter has more complete log
            if (self.getLogTerm() > lastLogTerm) or (self.getLogTerm() == lastLogTerm) and (self.getLogIndex() > lastLogIndex
                return (self.currentTerm, False)
            else:
                self.setVotedFor(candidateId)
                return (self.currentTerm, True)

        return (self.currentTerm, False)
```

5) The below two code snapshots show how the followers transition to being a candidate. And back to the followers if it fails to get elected as a leader.

```
    # Check if election HearbeatTimeout elapses. If so, convert to candidate.
    def updateFollower(self):
        if (self.startTime + self.electionTimeout) < time.time():
            self.initCandidate()
```

```
def initCandidate(self):
    self.state = 'Candidate'
    self.setCurrentTerm(self.currentTerm+1)
    self.electionTimeout = self.getElectionTimeout()
    self.startTime = time.time()
```

6) In the below code snapshot we are showing how we are sending messages in the form of JSON requests via UDP sockets in order to check the message communication between the nodes.

```
if __name__ == "__main__":
    sender = serverNode()


    # Creating Socket and binding it to the target container IP and port
    UDP_Socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

    # Bind the node to sender ip and port
    UDP_Socket.bind((sender, 5555))
    target = serverNode()

    # Sending 5 messages to Node 2
    for i in range(5):
        node_msg_bytes = sender.create_msg(i)
        UDP_Socket.sendto(node_msg_bytes, (target, 5555))
        time.sleep(1)

    #Starting thread 1
    threading.Thread(target=sender.sending_message()).start()

    #Starting thread 2
    threading.Thread(target=sender.listener, args=[UDP_Socket]).start()


    print("Started both functions, Sleeping on the main thread for 10 seconds now")
    time.sleep(10)
```

7) SafeLog Replication with Consistency Checks:

In the below screenshot we are trying to show when a client makes a request, how the request gets first copied to the leader nodes log and from there its further gets appended to the followers logs.

So basically we are trying to perform the below steps in the append entry functionality of the code:

a. The client's request gets appended to the leader's log

b. Based on the next index the leader sends the request to the followers by passing the entry(or request) in the AppendEntry RPC along with the prevLogIndex, prevLogTerm and leader's commitIndex.

```
appendEntriesRPC(self, term, leaderId, prevLogIndex, prevLogTerm, entries, leaderCommit):
    print ('Heartbeat ' + leaderId + ' ' + str(self.currentTerm) + ' ' + str(term) + ' ' + 
    if term < self.currentTerm:
        return (self.currenterm, False)
    elif term > self.currentTerm:
        self.initFollower(term)

    # Reset HearbeatTimeout
    self.electionTimeout = self.HearbeatTimeout()
    self.startTime = time.time()

    if prevLogIndex < len(self.log):
        entry = self.log[prevLogIndex]
        logTerm = entry["term"]

        if logTerm == prevLogTerm:
            self.log = self.log[:(prevLogIndex+1)] + entries

            if self.commitIndex < leaderCommit:
                self.commitIndex = leaderCommit

            return (self.currentTerm, True)
```

```
for i in range(0, len(self.nodes)):

    n = self.nodes[i]
    prevLogIndex = self.nextIndex[i]-1
    prevLogTerm = self.log[prevLogIndex]["term"]
    entries = self.log[self.nextIndex[i]:]

    try:
        socket.setdefaulttimeout(self.socketTimeout)
        t,v = n.appendEntriesRPC(self.currentTerm, self.id,
                                 prevLogIndex, prevLogTerm,
                                 entries, self.commitIndex)
        socket.setdefaulttimeout(None)

        if t > self.currentTerm:
            self.initFollower(t)
            break
        if v == True:
            self.nextIndex[i] = self.getLogIndex()+1
            commitCount += 1
        elif v == False:
            self.nextIndex[i] -= 1

    except Exception as e:
        print (e)
```

c. The follower will check to see if it needs to accept or reject the AppendEntry RPC by performing a log consistency check.

d. If the follower accepts, then it appends the entry to its own log and sends an APPEND_REPLY with success=True. The Leader on Receiving a True will update the next index and match index values.

e. Once the leader receives a reject from the follower, the leader will decrement the nextIndex[] for that particular follower by 1, and will send the previous log entry at that position back to the follower.

f. The leader will retry the AppendEntry RPC and eventually the we will reach a point

where the leader and follower's log match

8) Store and Retrieve:

Here we are trying to show the store and retrieve functionality.The below code is a snapshot from the controller.py which takes client the message in json format and passes on to the leader node.Here The STORE req appends an entry to the leader's log and this new entry gets sent along to the followers in the subsequent heartbeat/AppendEntryRPC.

```
# Socket Creation and Binding
skt = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
skt.bind((sender, port))
["CMD","app.py"]
# Send Message
try:
    # Encoding and sending the message
    skt.sendto(json.dumps(msg_sent['request']).encode('utf-8'), (target, port))
except:
    #  socket.gaierror: [Errno -3] would be thrown if target IP container does not exist or exits, write your listener
    print(f"ERROR WHILE SENDING REQUEST ACROSS : {traceback.format_exc()}")
```

The below screenshot shows how the leader after getting the client request updates its log entry and and also we are trying to show, when instead of a leader one of the followers sends a response to the client then it should respond with Leader Info the format of what constitutes a dictionary{ "Term": Term in which the entry was received, "Key": Key of the message ,"Value": Actual message (Could be some dummy value)

```
try:
    # Encoding and sending the message
    msg_received=skt.recv(json.load(msg_received['request']).encode('utf-8'), (receiver, port))
    Term=msg_received["Term"]
    Key=msg_received["Key"]
    Value=msg_received["Value"]
except:
    #  socket.gaierror: [Errno -3] would be thrown if target IP container does not exist or exits, write your listener
    print(f"ERROR WHILE RECEIVING REQUEST ACROSS : {traceback.format_exc()}")
```

```python
# Sending the client request to the leader node
def getLeader(nodes):
    leader = None
    for n in nodes:
        try:
            if n.isLeader():
                leader = n
                break
        except httplib.HTTPException:
            print ('HTTPException')
        except Exception:
            print ('Exception')
    return leader

# The leader node makes a log entry for the client request
def addEntry(nodes, tid, data):
    committed = False
    while committed == False:
        try:
            leader = getLeader(nodes)
            if leader is not None:
                committed = leader.addEntry(tid, data)
        except httplib.HTTPException:
            print ('HTTPException')
        except Exception:
            print ('Exception')
```

The below snapshot is taken from our node.py which indicates the listener thread at the controller end to receive messages from the leader.

```python
# Listener
# The controller can send a request to any of the nodes to RETRIEVE all the
# committed entries at that particular node.
def listener(skt):
    print(f"Starting Listener ")
    while True:
        try:
            msg, addr = skt.recvfrom(5555)
        except:
            print(f"ERROR while fetching from socket : {traceback.print_exc()}")

        # Decoding the Message received from Node
        decoded_msg = json.loads(msg.decode('utf-8'))
        print(f"Message Received : {decoded_msg} From : {addr}")

        if decoded_msg['counter'] >= 4:
            break

    print("Exiting Listener Function")
```

9) Append Reply:

When a follower receives an AppendEntry RPC (Also same for heartbeats), the follower would have the option to choose to accept or reject.

In this case the follower can do the followings:

a. Return false if term < currentTerm:

i.e, if the follower's own term is greater than the term mentioned in the

AppendEntryRPC, the follower rejects

b. Return false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm or if there is no term at prevLogIndex

c. If both the above conditions are not met, the the follower accepts the logs, appends any new entries to its own logs and sends an Append_reply with success=True

```
def appendEntriesRPC(self, term, leaderId, prevLogIndex, prevLogTerm, entries, leaderCommit):
        print ('Heartbeat ' + leaderId + ' ' + str(self.currentTerm) + ' ' + str(term) + ' ' + str(entries))
        if term < self.currentTerm:
            return (self.currenterm, False)
        elif term > self.currentTerm:
            self.initFollower(term)

        # Reset HearbeatTimeout
        self.electionTimeout = self.HearbeatTimeout()
        self.startTime = time.time()

        if prevLogIndex < len(self.log):
            entry = self.log[prevLogIndex]
            logTerm = entry["term"]

            if logTerm == prevLogTerm:
                self.log = self.log[:(prevLogIndex+1)] + entries

                if self.commitIndex < leaderCommit:
                    self.commitIndex = leaderCommit

                return (self.currentTerm, True)
```

Update the follower node's commitIndex based on the value sent by the leader and the length of the follower's logs

```
def execute(self):
    if (self.nextState != self.state) or (self.nextTerm != self.currentTerm):
        print ('Run ' + self.state + ' ' + str(self.currentTerm))
        self.nextState = self.state
        self.nextTerm = self.currentTerm

    if self.commitIndex > self.PrevIndex:
        self.PrevIndex += 1

    if self.state == 'Follower':
        self.updateFollower()
```

10) Leader Election (With log comparison to select leader):

The leader election will be similar to what we did in phase 3, but this time the follower node will have additional rules to grant positive vote:

Voting server will deny the vote if its own logs are more complete. I.e If:

(lastTerm$_V$ > lastTerm$_C$) OR (lastTerm$_V$ == lastTerm$_C$ && lastIndex$_V$ > lastIndex$_C$)

This ensures safety i.e. once an entry has been committed by a Leader no other node

```
def updateLeader(self):
    self.leaderLock.acquire()
    commitCount = 1
    for i in range(0, len(self.nodes)):
        n = self.nodes[i]
        prevLogIndex = self.nextIndex[i]-1
        prevLogTerm = self.log[prevLogIndex]["term"]
        entries = self.log[self.nextIndex[i]:]
        try:
            socket.setdefaulttimeout(self.socketTimeout)
            t,v = n.appendEntriesRPC(self.currentTerm, self.id,
                                     prevLogIndex, prevLogTerm,
                                     entries, self.commitIndex)
            x,u = n.requestVoteRPC(self, self.term, self.candidateId, self.lastLogIndex, self.lastLogTerm)
            socket.setdefaulttimeout(None)

            ################Leader Election (With log comparison to select leader)###################
            if t > self.currentTerm and ((t > x) or ( x == t and v > u )):
                self.initFollower(t)
                break
            if v == True:
                self.nextIndex[i] = self.getLogIndex()+1
                commitCount += 1
            elif v == False:
                self.nextIndex[i] -= 1

        except Exception as e:
            print (e)

    self.leaderLock.release()
    return commitCount
```

that doesn't have that entry gets to become the leader thereby preventing overwriting committed entries.

11) Integrate Client UI from Phase 1

The client that we have implemented with an UI in phase 1, is integrated with our RAFT cluster. This is done by having our client send a STORE request to the leader, where the value of the STORE request are the actual CRUD commands that the client is making.

12) Below is a snapshot of the docker compose file where we are trying to implement 5 nodes which correspond to 5 different containers. These containers are using the same image that is the raft application (node.py) with the same functionalities such as appendEntryRPC, requestVoteRPC, UpdateLeader, UpdateCandidate and UpdateFollower etc.

```yaml
version: "3.7"
services:
  node1:
    container_name: Node1
    build: Node/.

  node2:
    container_name: Node2
    build: Node/.

  node3:
      container_name: Node3
      build: Node/.

  node4:
      container_name: Node4
      build: Node/.

  node5:
      container_name: Node5
      build: Node/.

  #Do not change the part below
  controller:
      container_name: Controller
      build: Controller/.
      stdin_open: true
```

```dockerfile
FROM python:3.7-alpine3.14

ENV HOME /root
WORKDIR /app

COPY . .

EXPOSE 5555

ENTRYPOINT ["python3", "-u", "node.py"]
```

Difference between the basic log replication in the previous phase and the log replication in phase 4:

In the previous phase log entries were always empty as such client requests were not being made. However in this phase we have implemented this by taking the client request that we designed in phase 1, the controller.py sends this client request to the leader node , the leader then creates a log entry in its bucket for this client request and also replicates the same across all the followers.

Validation

1) We have done some simple testing by the
provided controller requests.

Testing Leader Elections: Raft application was tested running on 5 nodes and making sure the leader is elected properly. We have manually shut down the leader and rerun the application to check that another leader was correctly chosen. Then we tested restarting the node and made sure it was added back as a follower.

2) Further we have tested whether the candidate after losing the election transits back to being a follower on each of the five nodes separately, when the leader fails that is does not send heart beats within the time period.

3) We have also tested whether proper communication is happening between the nodes by encoding and decoding our request.

4) Safe log replication deciding who can and who cannot become the leader.

Reference:

1. https://cse312.com/static_files/slides/1_5_Docker.pdf
2. https://raft.github.io
3. https://raft.github.io/raft.pdf
4. Tutorial — Flask Documentation (2.0.x) (palletsprojects.com)