Distributed System Project Report
Siyuan Cao
6/3/2019

• The design and implementation of your distributed KV store including description of your interface and summary of how PUT and GET requests are handled.

In my program(udp_server.py), there are two classes. One class is called "State"(Line 8 – Line 21), which contain the variables mentioned in the Raft paper. All the variable names and its corresponding definition are exactly the same as Raft paper. Another class is called "cluster" (start from line 24), which includes all the methods I used to build this distributed system.

First of all, Line 25 – Line 52 includes all the variables that a node needs to initialize. Self.port use the current port number to bind a socket port in udp connection and also used as an ID number of the server. Self.total_port (Line 30 – Line 34) is an array that contains all the port of the system. It is calculated by the start_port and num_ports, which are command input by user. To simplify the user command input, I assume the user will use consecutive port number for the system. So if a user uses port 8000,8001,8002,8003,8004,8005 for the cluster, he only need to enter python udp_server.py 8000 8000 5 in the command line for server 8000. For server 8000, self.total_port includes all the ports except the port number of itself, which are 8001, 8002, 8003 and 8004. Self.storage (Line 37) is a dictionary to store the actual key-value pairs in each server sent from a client. Self.win (Line 40) calculates the majority number of servers that will be used in leader election and updating log replication commit. Self.leader (Line 43 - 46) indicates who is the current leader of the system. If there is only 1 server in the cluster, the leader would be the only started server, otherwise leader will be generated in the leader election section. Self.role (Line 48) stores the state of the server, it would be updated as state changes. 'F' means follower, 'C' means candidate and 'L' means leader. Self.recv_AppendEntries (Line 49) is a global variable used to update timeout when receive AppendEntries RPC from a leader. self.state (Line 52) uses the data structure of class State to maintain all the variables mentioned in Raft paper.

The main program (Line 414 – Line 428) takes port, start_port and num_of_servers as command line arguments and use these variables to initialize the cluster. After initializing the class object, the program will start a backend thread, "cluster_thread", that binds a socket port with udp protocol to listen all incoming requests. Cluter_serve method in Line 362 – Line 411 gives the definition of how a server listen socket would do. Line 363-Line366 creates a socket and bind to a port given by the first command argument. Line 368 – Line 411 keeps receiving message from either a client or other servers within the same cluster and send back corresponding reply. The listening udp socket receives a data in a format of byte string (Line 369). Python build-in Json library will read the incoming messages and load it into a Json format, which is similar to python dictionary (Line 370). When sending out message though udp socket, the program uses json.dumps(content).encode('utf-8') to convert a json object(content) into a string and encode into utf-8, then sends out the message. The listen socket only allows 4 different types of request, RequestVote RPC, AppendEntries RPC, put request and get request. All other requests are considered as illegal and return a fail message (Line 409 - 411). All request formats are defined as following:

RequstVo
te: Send:
{'type': 'RequestVote', 'payload': {'term':self.state.currentTerm, 'candidateId':self.port, 'lastLogIndex':lastLogIndex, 'LastLogTerm':LastLogTerm}}
Return:
{'type': 'RequestVoteReutrn', 'payload': {'voteGranted': True/False}}

AppendEntries:
Send:
{'type': 'AppendEntries', 'payload': {'term':self.state.currentTerm, 'leaderId':self.port, 'prevLogIndex':prevLogIndex, 'prevLogTerm':prevLogTerm, 'entries':entries, 'leaderCommit':self.state.commitIndex}}
Return:
{'type': 'AppendEntriesReturn', 'payload': {'success': True/False}}

PUT:
Send:
{‘type’: ‘put’, ‘payload’: {‘key’: ‘k’, ‘value’:
‘v’}} Return
{‘code’: ‘success’/’fail’}
GET:
Send:
{‘type’: ‘get’, ‘payload’: {‘key’: ‘k’}}
Returns:
{code: ‘success’/’fail’, `payload’: {‘message’: ‘optional’, ‘key’: ‘k’, ‘value’: ‘v’}}

Redirect Message:
{‘type’:’redirect’,’payload’:{‘port’:leaderID}}

So basically, the request server just sends out all parameters that RPC functions need to the receivers and specifies the request type. Once the receiver receives a message, it would distinguish the type of request first. If the type is ‘RequestVote’, it calls the method of self.RequestVoteRPC with all parameters received and return an output of the method as reply. If the type is ‘AppendEntries’, it updates the follower timeout, Line 384(used for heartbeats), and calls method self.AppendEntriesRPC with all the parameters received and return the output of method as reply. Put (Line 393-400) and get (Line 402-411) requests check the leader ID first. If a server is not the leader, it would return a redirect message with a correct leader port number that telling client who is the current leader of the system. Then, both will call a method and return the output of method as a reply message.

I will leave leader election and log replication in the next fault tolerant section. When receiving a PUT request from client, the leader will call PUT method (Line 318 - 337). The PUT method try to read the key and value pair in the message and append the pair into its log with the current term. Note that log is in a format of: [{key:value},term],[{key,value},term]…]. Each entry has a fixed index, an array index, and a term. Then the leader will send out an AppendEntriesRPC to all followers by calling method send_AppendEntries, Line 331. send_AppendEntries, Line 340-349, first issues one thread for each port. The number of threads issued are equal to the number of nodes in the cluster except the leader itself. Even if a node fails, the leader still sends out an RPC. Setting a socket timeout in the socket will take care of node failures. The leader keeps waiting for its commitIndex to be updated, then sends back a success, line 347-349. The GET method is quite simple. It just read the data from self.storage and return success and key-value if found the key or return fail if can't find the key, Line 352- 358.

Send_nodes method, Line 202-240, is called whenever sends out an AppendEntries RPC to a specific port. It has some fault-tolerant functionalities, but I write here because it is more of handling PUT requests. It is used in heartbeats and the process of handling client PUT requests. The method creates a new UDP socket port each time when calling it. UDP client socket is cheap and different each time, so it is ok to initialize a different one each time. The method then reads all the variables needed for an AppendEntries message, see AppendEntries format above, Line 207-211. prevLogIndex is the nextIndex of the port minus one. prevlogTerm just read the term of the entry on the log index. The entries to send is all entries in the leader's log starting at the nextIndex of port. It would an empty [] for heartbeats, because all followers already get the most recent entry and nextIndex of ports equals leader last log index plus one. Line 213 – 236 set up a socket timeout, sends out a message and waits for a reply. If a server crashes, the timeout would be caught by an exception, Line 238. If the reply from a server is True, the leader updates the matchIndex and nextIndex for the port. If the reply is False, the server decrements the nextIndex and checks if the server is still the leader, if yes retry the AppendEntries request, otherwise just stop running. A server might receive a heartbeat or AppendEntries RPC with higher term from another leader. If so, the server should be converted from leader to follower. This is a complex situation, but it could happen. For instance, if a follower's random timeout is 150 ms and another follower's timeout is 151 ms, the timeouts are so close. The first server converts to a candidate and sends out its ResquestVote but haven't receive anything back. The second server converts to a candidate and also sends out its RequestVote. Notice that they might at the different terms while sending out RequestVote. Both of them receive a majority votes and turn into leaders and sends out heartbeats. The server with lower term must convert into a follower from leader state. And the server with higher term rejects the lower term server. Line 161-164 also keeps checking if the server is still a leader.

- A summary of how your implementation is fault tolerant, including the fault-tolerant algorithm you have used.

All the servers are started as a follower, Line 48 and Line 428. It calls a Follower method, Line 55-77. The method has a random timeout as Raft paper mentioned between 150-300 ms. The method counts down time within a while loop. If receive a heartbeat or an AppendEntries RPC, it would update the timeout. If timeout gets to 0, the state of the server will convert to a candidate, increasing its term and calling the candidate method.

Line 80 – 127. A candidate votes for itself and starts an election timeout. Self.votes counts the number of votes the server gets for the current term. It sends out RequestVote RPC to all other servers, Line 88-96. It then starts a thread to receive replies from the other servers. In this way the main process won't be blocked by waiting for replies. Note the socket of sending out RequestVote is different from its self.port socket. The self.port socket listens for requests and call receiver methods, considered as a server. The RequestVote socket is used to collect votes, considered as a client. The thread starts a method receive_message, Line 130-142. Basically what it does is increasing 1 to self.votes when receive a Ture from a RequestVote request.Instead of listening in a while loop, it only listens for messages in the same times as the number total ports. Also, there is a socket timeout to deal with server failures so that the socket will stop automatically whenever pass the time, don't need to wait for a response from a crashed server. Back to the candidate method, Line 103 – 127, the while loop keeps checking the three conditions. If the candidate wins the majority votes, it would convert into a leader. If it receives an AppendEntries RPC, it would be a follower. If election timeout, it retries the candidate process again. Finally, a candidate will end up as a follower or a leader.

Line 244 – 277, ResquestVoteRPC method is called by a listen socket when receive a RequestVote request. This method has many restrictions to preserve consensus. Term is a very important factor in the cluster in order to keep consensus among servers. All servers should agree on the most recent term and turn into a follower if found out its term is outdated. It returns a False if the request term is older than its current term. It updates its term to the request's term if found out its term is outdated. Besides the term, another voting criteria is up-to-date of log status. Based on Raft paper, If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date. The request candidate should have at least as same up-to-date as the receiver candidate, which mean either the request candidate's last log term is greater than the receiver candidate's last log term or they have the same last log term but request candidate's log length is greater than or equal to the log length of receiver. After granting a vote, the receiver candidate will turn into a follower if the request candidate has a higher term.

Leader method, Line 146 – 190, is the core of the cluster. It keeps the log replication and handles all client requests. Self.lead_start time, Line 148, is used to count the timeout to send a heartbeat, I set it to 120ms, which is less than the timeout of any follower. When the current time passes the leader timeout, the leader sends a heartbeat message to every follower, Line 166-173. MatchIndex and NextIndex should be initialized whenever a new leader started, Line 150-156. Line 177-190 checks if there exist an N such that N > commitIndex for the leader's log. Because it is in a while loop, it keeps checking if the commitIndex is the most up-to-date. If a log is replicated on the majority of followers, the commitIndex would be updated and self.Update method, Line 193-198, will executes the storage process that stores key-value pairs into the state machine. This is related to Line 347 in PUT request. The leader replies to a client only when an entry is committed and applied to its state machine.

When a follower receives an AppendEntries request, it updates the follower timeout first then call AppendEntriesRPC method, Line 281-315. There are some restrictions that guarantee fault tolerant. First, the server returns False if receives a lower term RPC or it updates its term if receives a higher term. After term checking, the server will turn into a follower state and update the leader ID to the most recent leader within the cluster. What Line 296 – 304 does is matching the leader's log with the follower's log. The follower should append entries starting at the same last entry that Index and term matches the leader's log.

PrevLogIndex sent by the leader is based on NextIndex[port] variable stored on the leader side. If a follower returns a False, the leader would decrease NextIndex[port] and resend the request. If a follower has different entries from the leader, it would finally delete all the conflict entries and append all leader's entries to its log, Line 303 and 307. After a follower find out a log is committed, it would increase its commitIndex and executes the command in the log to update the state machine. After appending the new entries, the follower will return a True to the leader. Recall that a return of True will update the matchIndex on the leader side, Line 228. After reaching a majority matchIndex for log[N], commitedIndex of the leader will be updated to N. This ensures the correctness of log replication.

• A description of how you have tested your implementation to ensure that it meets all requirements.

1. Open 1 server. Send a client request and see if response is correct. Simple.

2. Open 5 servers on 5 terminals.

```
python 8000 8000 5
python 8001 8000 5
python 8002 8000 5
python 8003 8000 5
python 8004 8000 5
```

uncomment some print lines will show that a leader is elected as soon as 3 servers are started. Leader election, uncomment line 72, 106, 116, 125, 162 (162 is necessary because leader could convert to a follower). Heartbeats, uncomment line 167, 66 or 380. AppendEntries, uncomment line 329, 380.

Use udp_client.py to send out put and get requests to a random server. If the server is not a leader, it should return a redirect message. Then send requests to the leader. The leader should send back all the correct replies.

Cross out the leader terminal to make the leader crash. Remember to **cross out the terminal** not ctrl+C because there is a listening thread running in the backend. The cluster should generate a new leader and replicate log to all the followers.

Use udp_client.py again to send out requests, should test previous requests and some different requests. Receive all correct replies.

Cross out the second leader terminal. Now the cluster only contains three nodes. A new leader should be elected. Send client requests and should receive correct replies.

• Please remember to reference all sources used in developing your implementation.

I only use standard libraries. Also, all my implementations are based on the Raft paper. There was no other reference I used in my program.