Distributed System
Spring 2019
Yuqin Wu, Xiangda Bai

Project Report

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

First of all, we discussed and designed a communication architecture. The requirements are that each server should be able to talk to all other servers, while each server should be able to respond to all requests it received. In order to achieve these two requirements, at first, we thought we will need n ports and sockets for each server, in order to connect with all other servers, which will lead to n^2 ports and sockets. This is apparently not scalable. Therefore we look up all types of socket available in zeroMQ. We later found a forwarder mechanism. We almost adopted this approach, but later we discovered that, if the forwarder is dead, the entire system will be unavailable as well. This does not prevent any fault. It is not what we are looking for.

Later we designed a pretty easy-to-understand and scalable structure. We basically used a pub-sub pattern to implement the Raft algorithm. The key idea is that each server will manage a pub channel, with its own IP/port combination. All other servers will subscribe to this channel. Meanwhile, each server actually subscribes all channels except its own (because it is the publisher for its own channel). By doing this way, every server can send messages to all servers via its own pub channel (because all other servers have subscribed to this channel). Now, we meet the requirement for any server to send messages to all other servers. The second requirement is that any server should be able to send a reply of a request to the original requester. We solved this problem by designing a standard for the format of each message. There are basically two types of message for communication: one is request, and the other is reply.

We decided that for a request message, it must be sent in this format:
ALL,ALL,type(REQUEST/HEART/),content([phase1/phase2]&key&value),term,requesterIP,requesterPort
Each field is separated by a comma. The first two are IP and Port field. Since in leader election, requests are always sent to all other servers, the first two fields will be set as "ALL" to indicating all servers who see this message should receive it and handle it. The third field is the type of message. "REQUEST" is when a candidate sends a vote request to all other servers requesting for a vote. "HEART" is the heartbeat message, which is used when leader server regularly sends heartbeat messages to all followers. "content" field is used when leader receives an update request from clients, the log replication algorithm will make use of it. The format of the content is that, regularly, it is just an empty string. If there is an update that needs to be broadcasted, the content will be filled as phase1&key&value. The first one is indicating the current phase. For phase 1, all followers who receive this message will send back a reply to confirm the log this update, we will talk about this in detail when we reach reply format explanation. When it is

phase2, followers will commit this change and send back a reply to confirm. The last three fields are straightforward, it is the current for this message and the address of the request sender.

For a reply message, its format is:
requesterIP,requesterPort,type(VOTE/ACK/READY/COMMITTED),content,term
The first two fields are the requester address received from the request. The third field is the type of message. “VOTE” is the reply for a candidate’s vote request. “ACK” is the reply for the heartbeat message. “READY” is the acknowledgment for the heartbeat message with phase1 content. The last one, “COMMITTED”, literally is the acknowledgment for the heartbeat message with phase2 content.

Now you understand the format of all messages exchanged. We start to explain how messages are handled. There are two main rules:

1. For each subscriber, it only receives messages begin with two “ALL” fields (this is the case for receiving requests that aim to send to all servers) or the first two fields match the receiver’s IP-port address (this is the case for receiving other server’s replies to its own request previously sent).
2. For each server, no matter which role, follower, candidate, or leader, it currently is, it only receives messages with the term equal or larger than its own term.

All other messages do not meet this requirement will be filtered out.

After we figured out the message exchanging mechanism, we start to figure out how to deploy it into python code while preserving object-oriented principles. I made three main classes. One is the basic class used to store the configuration info of the server. Data like its own IP, pub-port, number of total servers, its current role, the leader's address, and so on, will be kept in this class. The second class is the subscriber, it is a thread class. In order to receive and send messages concurrently, we decided to spawn threads for each channel that needs to be subscribed. For example, if there are 5 servers in total, then for each server, it will have one pub channel, and 4 threads subscribing all other channels, we call them subscribers. There is a shared message queue among subscribers and the main thread. Each time a message, which meets the above requirements, will be put into this shared queue. There are 3 main functions for main thread: follower(), candidate(), and leader() all together represent different states for a server. Main thread handles the state transition for different roles. Inside these three functions, they will send their corresponding messages and read and handle the messages from the shared queue, based on the leader election algorithm. The third class is a client socket class. This is also designed to use thread, so we will spawn another thread, which we called client thread, in the main at the beginning. This thread solely interacts with clients. It operates a reply socket binding the outgoing port, which is open to clients. It handles client requests, sends the corresponding message on another shared queue, which we call it LogList, and make a response after an answer is out, or reply failed after a timeout.

There are two types of requests, “GET” and “PUT”, a client can send to any server by connecting the outgoing client port. The client thread class handle these two types of requests received from clients. For get-type message, it will read the local copy and reply with the result. There is no synchronization needed. For put-type message, it will read the current role. If it is a leader, it will initiate the log replication algorithm. If it is a follower, it will redirect the request to the leader, and receive a response from the leader, and redirect the response back to corresponding clients. If it is a candidate, it means there is no leader yet. It simply replies a failed message.

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

We basically use the Raft algorithm to support our fault tolerance. Our leader election algorithm is able to handle possible faults.

First, in a running system with one leader and many followers, the leader goes down. In this situation, followers will keep counting down until the set timeout is reached. A follower will become a candidate after the timeout and send a vote request to all other servers. Meanwhile, there may be other candidates sending vote requests so the candidate server receiving the majority of votes will become the new leader. The new leader will send out heartbeat message to all other servers. The other candidates or followers will know there is a new leader and keep running as a follower.

Second, a follower is down. In this situation, the leader and other followers will keep running. In the next possible voting round, a candidate can be a leader when receiving the majority of votes. There can be a problem when more than half of the servers go down but this situation is not in our consideration.

There also can be some situations in which message transfer takes too long and servers will send or receive some message that not relevant to its current state(follower, candidate, leader). For example, a leader goes down for a period of time and during this period, a new leader is elected. When this leader comes back, its state is still the leader and it will send out heartbeat message to all other servers including the new leader. So we added a term variable to all messages in the transfer. When a follower becomes a candidate, the term goes up. When a server receives messages with a lower term, it will ignore this message. If a server receives a higher term message, the server will update its term and change state accordingly. When this server receives a message with the same term of itself, the server will do the proper action. So in the fault example above, all other servers including the new leader will ignore the heartbeat message from the recovered leader and this old leader will become a follower after receiving the higher term heartbeat message from the new leader.

On the client side, all failures happen on the server side will just return a fail message instead of crashing the client.

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

Leader Election Tests
Test 1. Start three servers and one leader is elected. Kill the leader to see if a new leader is elected.
Test 2. Start three servers and one leader is elected. Kill a follower to see if the system can keep running.

Test3. Set the heartbeat time longer than follower timeout. Start three servers and one leader is elected. Followers will not receive second heartbeat message before timeout so one or some of them will become a candidate with a higher term. See if a new leader can be elected and the old leader become a follower.

KV Storage Tests

Test1.

Start three servers and connect a client to the leader.

Put a KV pair into the system. Get the value for saved K.

Kill the leader.

Connect the client to a different port of the system.

Get the value for saved key.

Save a new KV pair into system.

Get the value for new key.

Test2.

Start three servers and connect a client to a follower.

Do the same test as in test1.

Test3.

Start three servers with heartbeat time longer than server timeout. Leader will keep changing among servers. Connect a client to random port of the system.

Do the same test as in test1.

Test4.

Start one server and connect the client to it.

Put a KV pair.

Get value for saved key.

Reference:

1.ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm (Extended Version). Stanford University.

2.Raft Understandable Distributed Consensus. http://thesecretlivesofdata.com/raft/

3. ZMQ Publish/Subscribe Pattern.

https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/pubsub.html