Distributed Systems
Final Project Report

Winshen Liu and Aaron Tinn

6/3/2019

Design, Implementation and Background

Design

Our distributed key value store is based on the *Raft Consensus Algorithm*. This allows a client to submit requests to a system of five servers that provide for fault tolerance and prevent conflicting operations.

Raft achieves consensus through coordination by an elected leader. A server in a Raft cluster can take on one of three roles at a time: leader, follower, or candidate. The leader is responsible for handling client requests and directs followers to replicate log entries and commit to their state machines. To maintain power, the leader regularly sends a heartbeat message to its followers.

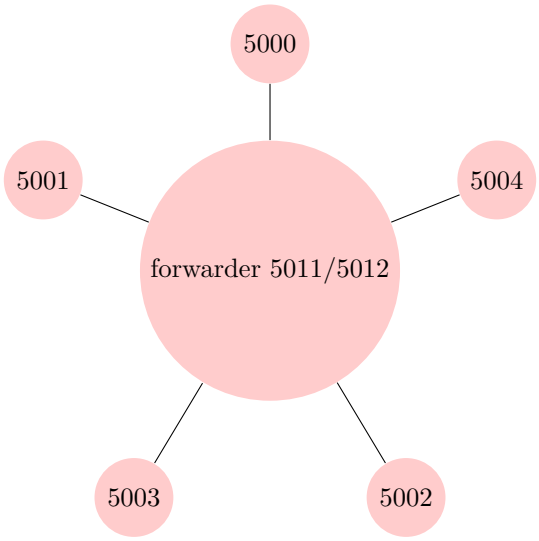
The follower is responsible for responding to the leader's direction on updating log entries and state machines, as well as running for election in the event that a leader is unavailable and voting for candidates in an election. All servers begin as a follower. Each follower has a randomized "election timeout" (typically between 150 and 300 ms) in which it expects to receive a heartbeat from the leader. The timeout is reset upon receiving the heartbeat. If no heartbeat is received before the timeout expires, the follower changes its status to candidate and starts an election for a new leader.

The candidate is responsible for requesting votes from followers and becoming the leader upon receiving a majority of votes. In the event there is more than one candidate running for election at the exact same time, our implementation of Raft sets the candidates back to follower status with randomized timeouts. Such randomization should prevent several split votes in succession. Upon receiving a majority of votes, the candidate becomes a leader and begins sending heartbeat messages. To prevent new elections, note the leader's time interval for sending out a heartbeat message must be more frequent than the followers' election timeout interval (i.e., less than 150 ms).

Implementation

We implemented this Raft-based distributed key-value store across five servers using Python 3.7 and the ZeroMQ library. Our main communication between the servers uses the PUB/SUB model in ZMQ by publishing messages to a forwarder and subscribing messages from a forwarder so that every server can publish and subscribe to each other, including themselves. Communication between clients and the servers is implemented with ZMQ's REQ/REP model.

Below is a model of the server clusters (Ports 5000 - 5004) and a forwarder device that sends listens on 5011 and distributes on 5012:



(1) To start our system, the forwarder is launched. Then, the servers are launched, during which they elect a leader. Then the client starts and the user is prompted to connect to a port that corresponds with one of the servers in the system. Once a port is selected, the client is connected to the selected server through a REQ/REP channel. The client then enters either a "PUT" or "GET" request.

(2) For "PUT" requests, the client enters a key and value and for "GET" requests, the client enters a key to search. Regardless of the operation, the request is sent to the server over the REQ/REP channel.

(3) Once the server side receives a client request, the request is processed by the leader. If the connected server is not the leader, the client is provided the leader's port number and reconnected to the leader's port. The original request is redirected through this connection and the client is made aware of the redirected connection.
(4) The request is then processed by the leader, who stores the entry in its log and signals to its followers to append the new entry in their respective logs.

For "PUT" requests, the leader then commits the entry to their local database, a Python dictionary object. For "GET" requests, the server looks up the key in its dictionary and returns the value if the key is found or a "not found" message if the key is not found.

(5) The server sends a reply through the same REQ/REP channel with a status message of its success or failure in completing the operation.

(6) After each operation, the client is prompted to submit another request. If the server they were connected to is no longer alive, they are prompted to select another port. Their subsequent requests proceed as described above in steps (2)-(5).

To make the above functionality possible, each server is parallelized to run four threads: (1) Client request thread - listens for client requests (note this has activity only when the server is a leader) (2) Heartbeat thread - listens for and responds to heartbeats and append entry messages from/to the leader (3) Election thread - listens for requests to vote and responds with votes from/to the candidate (4) Timeout thread - counts down a randomized amount of time to manage election timeouts

Fault Tolerance and Safety

Our Raft implementation handles server failures and guarantees basic safety as detailed below:

(1) Node failure When a follower node crashes, the system can continue responding to client requests without change. All requests sent to the crashed node are ignored. Though the node no longer sends or receives messages, the other nodes are unaffected and can proceed without the node. This is because the algorithm operates on the majority of live servers consenting to a given operation (e.g., log update, leader election).

When the leader node crashes, the followers are made aware by the lack of heartbeat messages. The first follower to exit their timeout without having received a heartbeat will start a new election. It will increase its term and become a candidate, then vote for itself and request votes from other servers. The remaining follower nodes respond to the request for votes with a vote for the candidate. Once a majority of nodes have voted, the candidate is elected leader and sends heartbeats.

If a client sends a request to a crashed node, they are notified the server is no longer available and to try connecting to another port. If a client sends a request to a node during an election, the client is notified to retry later. However, since the election completes in a matter of seconds, this scenario is rare.

We chose to use the forwarder model, similar to dealer/router models, in ZMQ to facilitate communication between servers. However, our design does not withstand failure of the forwarder. If the forwarder were to be killed, the servers would not be able to properly send and receive messages on the PUB/SUB channel, preventing effective coordination and consensus. This is a limitation of our design but one that allowed for streamlined server communication.

(2) Election Safety: At most one leader can be elected for each term. A candidate will only send a request for votes once per term/election. Each follower will only vote for one candidate per term and since a candidate must receive a majority of votes in order to be elected, only one leader can be elected for the given term.

If two or more candidates run for election at the same time, neither will receive a majority of votes. Without the majority after some time, determined with a separate randomized timeout, they are reset back to followers and subject to randomized election timeouts again. This should result in a single follower becoming candidate first, which allows a single leader to be elected.

The probability of having another split vote exists, but is small given the randomized timeouts. Even if a series of split votes were to take place, the algorithm is able to start new elections until a leader is established.

(3) Request Operation Safety: Our implementation ensures there are not conflicting requests on the same value through the assumption of fail-stop failures and using a queue and leader to manage requests.

Given our implementation only needed to work for fail-stop failures, meaning nodes do not recover from failures, there is no way for logs to have conflicting entries. If multiple logs have an entry with the same index and term, then those logs are guaranteed to be identical in all entries up through to the given index. A follower will not vote for a candidate whose log is less up-to-date (i.e., candidate's last log index is lower than the follower's.)

If a client submits a request to read or write a value, it will not conflict with any other operations because only one server (the leader) processes the request and the leader can only append new commands to its log. If multiple clients submitted requests, ZMQ organizes them into a queue and they are handled by the leader in order. Thus, multiple "PUT" and "GET" requests would not result in conflicts.

For a given request, a single leader server can only append new commands to its log. Deleting is not permitted. If a server has applied a particular log entry to its state machine, then no other server in the server cluster can apply a different command for the same log index. Once it has updated its own log, the leader directs all other servers to replicate the log entry. Followers return a consensus message letting the leader know when they have added the log entry. Only upon receiving a majority consensus does the leader commit the entry to its database and instruct followers to do the same. If no majority is reached, the leader

does not commit the entry and the entry is not added to any state machines in the system.

Testing

We conducted testing in three phases: single server and single client testing to test key-value storage, server-side testing to test election safety, and end-to-end testing to test fault-tolerance.

Key-Value Store Testing
The first piece we tested was ensuring a single client and server could handle "PUT" and "GET" requests. We started one server instance and one client:

- **Put key-value:** The client submitted a "PUT" request with a key ("hello") and value ("world"). We checked the server by printing out its database and ensuring the key and value were stored. We then checked that the client received a status message back (e.g., "Success").
- **Get non-existing key:** The client submitted a "GET" request for a key that was not in the server's database. Upon starting the application, we immediately tried a "GET" request (e.g., "mpcs"). Since there had been no preceding "PUT" requests, this "GET" request can only fail. We ensured the server side did not have such a message in its database and returned a "Success" status with a "not found" value to the client.
- **Get actual key:** The client submitted a few "PUT" requests, then the client submitted a "GET" request of a key we entered previously. For instance, the client would send the following requests in order: PUT: key: hello, value: world =¿ Server reply = "Success" PUT: key: hola, value: adios =¿ Server reply = "Success" GET: key: bonjour =¿ Server reply = "Success, value not found" to indicate the "GET" request was successfully executed on the server, but no key was found and thus there is no value for the key requested

Server-Side Testing
Our code was then extended to handle multiple servers and implement parts of the Raft algorithm. We tested the following by running the forwarder device and 5 instances of our server (without any clients):

- **Initial leader election:** To ensure there was only one leader at a time, we ran the server-side of the system and printed out statements counting down each server's election timeout, if they ran for election, how they voted, and who the current leader was. We also increased the timeout windows (e.g., to be between 3 and 10 seconds) so there would be sufficient time for us to check the state and progress of each server and check there was only a single leader in power at a time.

  Assertions: Each server prints out their election timeouts and the timer counts down from an initial start between 1.5 and 5 seconds. If no heartbeat is received in that time, the server prints out a request for votes to reflect the message it sent to followers to start an election. Followers print the candidate's last log index compared to their own and ensure they match before printing the server ID of the server for which they vote. Upon receiving a majority of votes, the elected leader prints heartbeat messages to the terminal each time it sends a heartbeat to followers.

- **Leader election after failures:** We conducted the initial leader election test described above, then forcibly killed the leader node (Ctrl+c).

  Assertions: Live servers do not receive heartbeats and therefore their election timeouts continue counting down. The first server to have their timer run out sends a request for votes. The other servers vote for the candidate and the candidate, upon receiving a majority of votes, begins sending out heartbeats. We repeated this so that only three nodes remained and ensured one and only one leader was elected after each failure.

- **Split votes:** We reduced the election timeout window and reran the server side multiple times to force a split vote. Upon seeing two or more servers send a request for votes, we checked that candidates would restart a new timer as a follower and if no heartbeats were received before the timer ran out, start another election.

  Assertions: After a split vote, the candidate servers revert to being followers and their timeouts count down. The first follower to have their timeout expire becomes a candidate and sends a request to vote. The rest proceeds as expected for election of a single candidate.

End-to-End Testing

Once server-side testing was completed, we started multiple clients, the forwarder, and all five servers (ports 5000-5004). We then connected clients to an arbitrary server from the list of possible servers and sent "PUT" and "GET" requests to update and retrieve values from the database without any knowledge of the backend. This would ensure total transparency in the event of failures. The following was tested:

- **Client redirects:** We intentionally connected the client to a follower when there was a leader in power then submitted "PUT" or "GET" requests to said follower.

  Assertions: The client receives a "redirect" notice along with the port of the leader and is automatically reconnected to the leader, with its original request now submitted to the leader. The leader processes the request accordingly.

- **Client reconnects after leader deaths:** When the client is connected to the leader, we kill the leader.

  Assertions: The client receives a notification that the server is no longer available and is prompted to try another port.

- **Client retries during elections:** We extended the election timeout window and connected the client during the initial leader election.

  Assertions: The client receives a message to try back again (after the election is complete). After waiting a few seconds, by which point a leader would be elected, the client submits requests and they are processed as expected.

- **Server log replication:** We set each server to print their log each time an entry was added to their logs. The client sent several "PUT" and "GET" requests.

  Assertions: The client received successful replies and the servers' logs match the client's requests in order as well as the leader's log.

- **State machine replication:** In addition to printing their logs, each server was extended to print their state machines. The client sent several "PUT" and "GET" requests and received success replies. Then, the leader is killed.

  Assertions: The client is prompted to connect to a different port. Upon connecting to a new port (which may be a follower or the new leader), the client is able to successfully "GET" requests submitted to the database in previous "PUT" requests (to the now-dead leader). The client is also able to successfully "PUT" requests to the new leader. This remains true after another leader failure.

Sources

Raft paper: https://raft.github.io/raft.pdf
Forwarder: https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/devices/forwarder.html