CPEN 533 Report
PA3: fs533

Kexin Wen (76020940)
Jing Cao (62040092)

Overview
In this project, our goal is to implement a simplified distributed file system called fs533.
All the basic functions are implemented based on the paper of Raft. We have successfully implemented consensus, file management which is using metadata. In addition, we have implemented metadata replication.

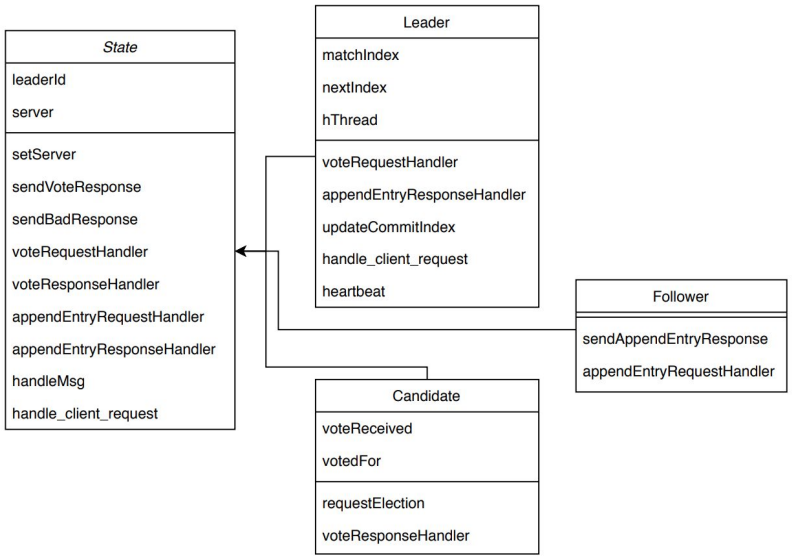
Design rationale
To build the distributed file system, instead of implementing it based on the Hadoop Distributed File System, our design is based on the paper that introduces Raft algorithm [1]. Raft is easier to implement compared with other consensus mechanisms, and it can reach consensus in distributed systems so that our file system will function as long as more than half nodes are working.

System structure
The clients can interact with the system by having several requests to the leader server. Each client has an id, an ip address and a designated port number for the first request. Each server has three threads, one for processing client messages, one for subscribing and another for publishing tasks from other nodes. There is one leader and multiple followers in the system. The elected leader handles all client requests and manages the system i.e consistency, replica, metadata.

Design Component
Server state
Based on the Raft mechanism, every server would hold one of the following states: Leader, Candidate or Follower. All these classes are extended from the basic State class. All the communication between each node will be handled by messaging. Thus, each role would have its own methods to handle different types of messages.

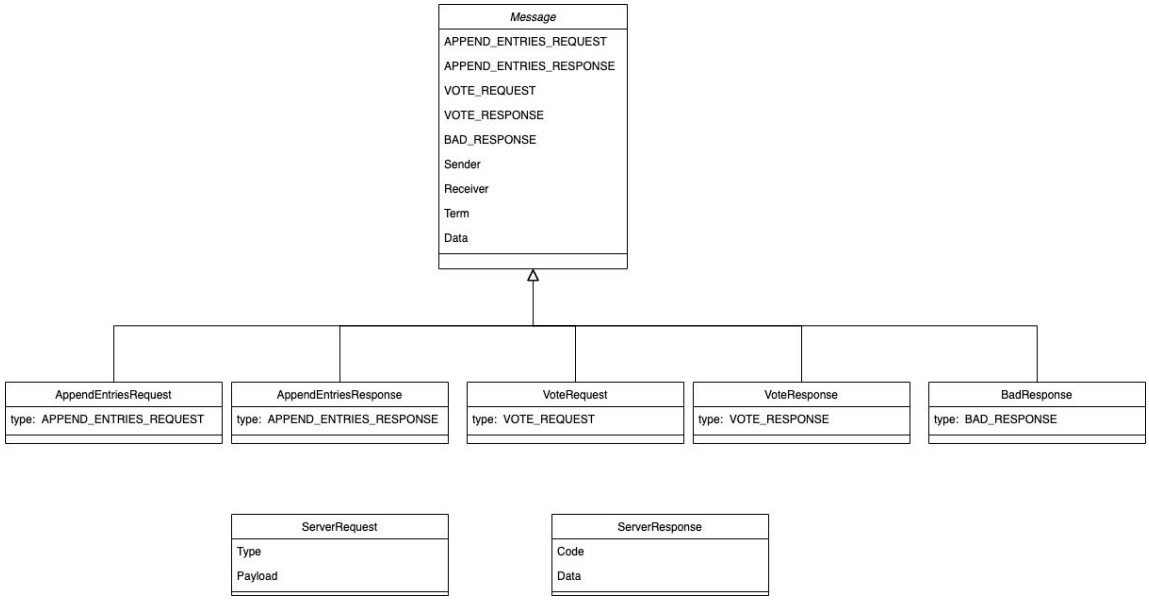
All the servers would start as a follower state. Each server can change its state among the above roles when there is a timeout. We used a randomized election timeout to accelerate resolving the split vote problem. Once the current leader fails or at the initialization step, there would be a leader election among the servers. The follower would become a candidate if it does not receive heartbeats from the leader over the pre-defined duration. To begin an election, the term counter incremented by one, indicating a new term starts in this system. Each candidate votes for itself and sends a <VoteRequest> message to all other nodes. The candidate receiving a majority vote would become leader. If a leader or candidate receives a higher term id from another server, it would change into follower again.

Each server has its own current term counter which will be attached in each message. Servers with a higher term number will reject messages from servers with lower term numbers. Servers with lower term numbers will receive messages from servers with higher term numbers and update their outdated term number.



Message system

We implemented the communication system by using ZMQ, which is an asynchronous messaging library easy to apply. For inter-nodes communications, five types of messages are created that are derived from the basic Message class. There are two other types of messages (ServerRequest, ServerResponse) designed specifically to communicate between server and the clients.



Replication strategy

Log replication

Once elected, the leader would send periodic heartbeat messages <AppendEntriesRequest> using TCP to all the followers in order to maintain its authority. When the new metadata modification request is appended in the leader's log, the leader will forward the log entry as <AppendEntriesRequest> to the followers. After the leader receives confirmation from the majority of followers <AppendEntriesResponse>, the log entry has been replicated and the request is committed. This commit also includes all the leader's previous preceding entries. Now the leader will append the highest committed index to all the messages. Once the followers receive the message from the leader, they learn the log entry has been committed, so they will also commit their corresponding entry. This two phase commit ensures log consistency among all the servers.

The leader contains a dictionary called 'nextIndex' for each follower. It holds the indexes of the next log entry to send to that server. When the follower has a divergence after receiving a message from the leader, the 'nextIndex' would not match. So the follower will decrement the nextIndex and retry until the corresponding nextIndex is matched, then the follower would append the rest of entries from the leader's log. This ensures synchronization between followers and the leader to achieve the correctness of the data stored.

Fault tolerance

For servers within the cluster, when the process crashed, the leader would change. But the new leader has all the committed log entries of the previous leader. When the follower or candidate crashes, the Raft algorithm would still allow other servers to try to send <VoteRequest> and <AppendEntriesRequest> to the crashed one. Since the message is idempotent, receiving multiple copies would have a single result.

For metadata recovery, the leader manages metadata and metadata has replications on all nodes. Once the leader fails, the new elected leader will have an up-to-date metadata replication which ensures its fault tolerance. Since metadata on each node maintains its consistency by the Raft consensus.

Client-server communication

The client can send requests to any node in the system and if the destination node is not the leader node, the client will be redirected to the leader node. The leader node can then process all the requests and commit logs.

Future work

In our design, chunks must be split by size on the client side and sent to servers to store with respect to the instructions from the leader. However, due to time limitation, we have not accomplished this part. In addition, we haven't implemented merging chunks into a whole file. Furthermore, we could implement file transfer in the future.

References

[1] Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. In 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14) (pp. 305-319).