Distributed Systems - Coursework Report

Neel Amonkar - S2030247, Anthos Makris - S2036401, Anish Thapa - S2024006

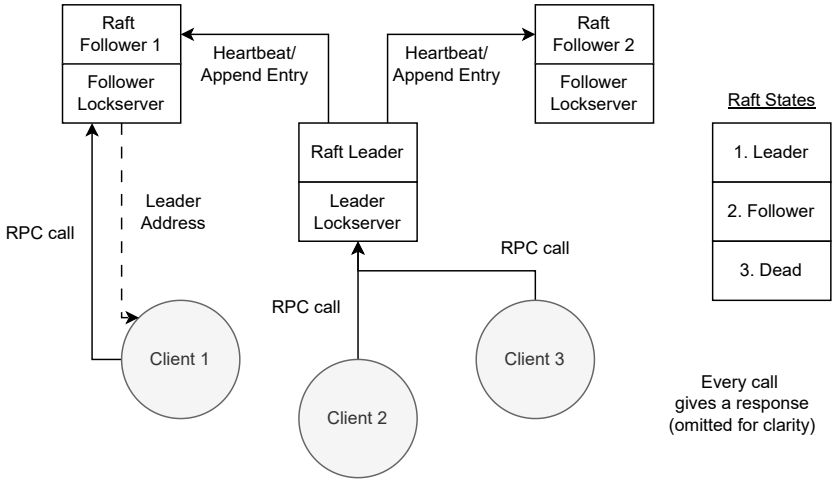

Figure 1: System architecture diagram; here, Client 1 communicates to a follower and gets a response that points to the leader server. Clients 2 & 3 communicate as normal.

*Disclaimer: We did not use Raft in our system, but Raft persisted as a name for our replication/consensus nodes.

Assumptions

We assume that the client only sends requests one at a time, and that the clients are single-threaded. We also assume no malicious server nodes. Currently, the servers run on separate ports on the same machine; we thus assume low latency communication. Since the clients also run on the same machine, the server cannot assign unique IDs depending on the client's IP address, therefore the client provides a unique number during RPC_client_init. We simulate lost packets using a flag in RPC calls, where the client either does not send the call (packet lost before execution), or drops it without updating the state (lost after execution).

Goal 1 - Client-side retry

The client class contains an internal retry mechanism, invisible to an external user of the class. If an RPC call fails, the client automatically retries the call. In a multi-server setup, we handle the case where a client accidentally contacts a follower – the server returns a response redirecting it to the current leader, and the client continues retrying there.

Goal 2 - Server handling duplicate requests

The server keeps track of the expected number of the next RPC call for each individual client. Only valid calls cause an increase of this sequence number, both on the client and the server sides – in case of a mismatch, the server sends a sequence error. Sequence numbers are abstracted away from the user, and are used to handle lost RPC calls on the client side.

Goal 3 - Lock waiting list

To keep track of the clients waiting to acquire the lock, we use a Python deque as a waiting list. When a client calls lock_acquire, its ID number gets added to the end of the queue. The server thread handling each client's lock_acquire blocks until the client becomes the lock holder (spinlock). This ensures that the client cannot call any further operations until it receives a positive response from lock_acquire.

We also define a lock timer to ensure no client holds the lock too long. The timer is reset every single time the server receives a file_append call from the current lock holder.

Goal 4 - Lock holder safety

The lock-server manages the lock holder by keeping track of the client currently holding the lock by ID – only the lock_owner client is allowed to append to files or release their lock.

We also implement transactional behaviour for file changes. The server has another Python deque to keep track of the current lock holder's file_append calls, which are executed in sequence when it calls lock_release. This deque is reset (without execution) if the lock timer expires and the lock is passed to the next waiting client.

Goal 5 - Logging and server state recovery

We define 5 command type classes, for every possible change to the internal state: adding a client, incrementing a client's sequence number, changing the lock holder, adding a file append operation to the deque, and executing all queued appends. The full waiting list state is not preserved, however.

The server maintains an internal log of LogEntry objects (each containing one Command), and serialises to a JSON. Upon revival (assuming a one-server setup), the server deserialises this JSON and executes all the commands to recover the previous state.

Replication

- We use a strong consistency model – when the leader receives a call from the client, it sends the appropriate command to the followers and waits for them to respond before it processes it itself.
- An exception is when a server dies; the lead server can kick a dead follower out of the group, instead of waiting for it to come back.
- Upon revival, a server assumes it is out of date. As such, any server recovers as a follower – it searches for the new leader, recovers any missing log entries, and "subscribes" to them. If it doesn't find a leader, it starts its leader timer.
- The lead server stops processing new RPC calls until the follower's recovery is complete. This ensures that all servers contain the latest state. As well, the follower does not handle any client calls it happens to receive during recovery.
- If the leader server dies, any of the followers can take its place, since we assume all non-recovering followers are up-to-date. Each follower server has a randomised timeout, for when no messages from the leader are received – the follower with the shorter timeout becomes the new leader.
- The leader holds a list of subscribed followers to which it sends all RPC calls as well as heartbeat messages. The server maintains a thread for heartbeats – the interval here is shorter than the lower bound for the follower timeout, so followers cannot conclude the leader is dead too early.

Limitations

- As our model is strict (though with timeouts), our system is not exactly scalable.
- Recovery is slow, and is only guaranteed to be correct if an alive leader exists.
- We assume that any up-to-date followers become the leader before a reviving follower. However, if all servers crash, an out-of-date server may revive before more up-to-date nodes, not update itself, and become the new leader by chance.
- As the waiting list is not preserved in full, there are fairness issues when a server dies and comes back. In any case, the client's earlier lock_acquire call would fail when the server dies, so the client would need to call the function again.