CS651 Distributed System
Rui Dai
Wenjun Shen

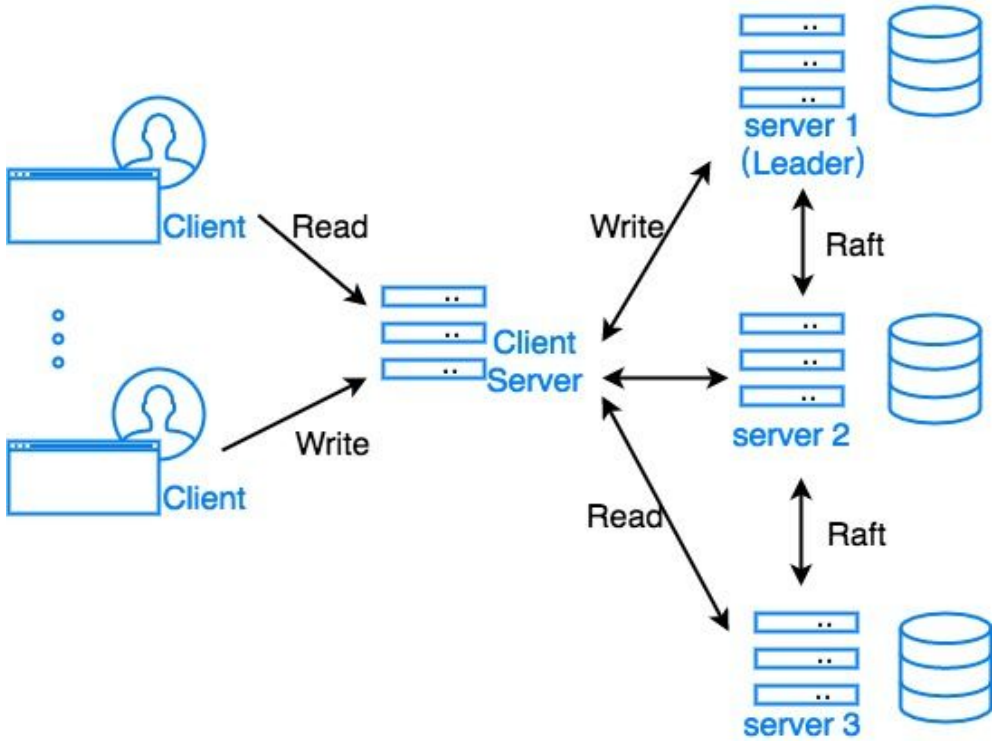
Final Report

Project Target

To build a distributed, centralized, fault-tolerant website. There are multiple server running for the website. If one of the servers is down, the other servers will still make the website reachable to users. The function of the website is similar to Reddit, user can login, view links and post links to the website. To the client's' point of view, the website makes no difference no matter which server they are connected to - if clients are logged in to a server which latter downs, the substitute leader should also show this status.

Project design

The basic structure of our website is showed in picture below:



The basic process of visiting our website is as follow:

1. Client makes a request, either a simple GET(Read), POST(Write) or LOGIN to our website.
2. Our Client Server reads the request. If it's only a GET, it fetches a server's IP address then forward client's request to this server. If the request is POST or LOGIN, it finds the current leader and forwards the request to the leader server.
3. a. If the request was GET, the server returns data to client server and client server return the web homepage to client.

   b.If the request was POST or LOGIN, Leader server first log user's request then replicate it with all the other servers, then responds to client server and client server return corresponding web page to client.

In our design, there are multiple servers running for the website. Each server communicates to each other through rpc. To make it consistent, servers will store user's' requests locally and replicate logs by raft protocol. We divide our clients' requests into two types: read and write. We optimize our website by using eventual consistency on read (GET) requests. Website user may read stale content in website if new post are not updated. But this window is short because the new post will finally be updated within less than a few milliseconds. For all the read requests, client server will forward them to any server in the system(may be the geographically closest one). For write requests, client server will only forward it to leader server. Leader server will first log this write request and let raft handle the log replication. This design makes our website fault-tolerant, because client server machine will always give a reachable IP address. And that all modifications to the website (ie. posting links and logging in) would be consistent.

This design reaches a good load balance and avoid the latency caused by raft since read request won't be log and replicated.

Implementation:

Raft:

Since raft uses librpc to simulate network communication, we modify raft so that it is able to communicate with each peers through actual network by using http/rpc package.

Basic Functionality:

Website supports sign up, login, post link and give likes to link. (similar function to Reddit). User can visit all the link in our website and post link when logged in.

Concurrency problem:

We use rpc as connection between leader and followers, and between servers and client server. However, the golang/rpc can not listen to different rpc endpoints on same server. We use the solution online by customizing rpc server. [1]

Test setting:

During our experiment, we use three laptops, and each one is considered as an individual server. Each server listens to port:8000, which receives requests from client server.

Each server has an underlying raft, which listens to port:8080 that communicates between raft peers. We set up client server on csa2 server, it listens and serves on port 8000. When a client visits 128.197.11.36:8000, the http.Request is parsed by client server and encoded into rpc request to the leader by client server. After receiving reply by leader server, client server encodeS the data in http.WriteResponse and sends it back to client. We write test cases in kvraft/test_test.go and dns/test_test.go that initialize Servers and Client server.

Test Case1: Leader server is shut down.

Users are still able to access website and perform all operations. User login state remains consistent. After user successfully logins, and leader server is shut down, the

new leader will still have the user's login information. User stays login even switching to another server.

Test Case2 : Old leader server comes back after a shutdown

After the old leader server rejoins the network, it hears heartbeat from new leader and becomes a follower. This server will update its local log if any new log is received from new leader. User won't notice when the old leader server rejoins.

Test Case3 : Follower server is shut down

The website is still reachable and workable. Client server will know this server is down and no longer forward Get (Read) request to this server. User won't notice when a follower server is down.

Test Case4 : Follower server comes back after a shutdown

After this follower server rejoin the the network, it hears heartbeat from leader and update its local log if any new log is received. User won't notice when a follower server comes back.

Future Work:

When all servers are down, data will be lost because we do not write data to disk in the current model. To maintain persistent, we can modify the scheme by storing data locally to disk so that data is stored on disk when a server is down. We implement a Client Server to help redirect client requests to servers and server response to client. This is a bottleneck since all the messages have to go through the Client Server. To reduce Client Server workload, we can try to implement the design that server reply to the client directly. Meanwhile, this Client Server is necessary because clients do not know who's the current leader. However, this implementation requires this Client server never downs - otherwise all connections to the website are not reachable. In the future we could find a way to make the real world DNS help us redirect client requests - since we have not registered for a real domain name for our website, this approach was

discarded for our current implementation. Furthermore, we hope the redirection could be optimized by identifying the geologically closest server to the client to serve all get requests, thus reduce the lag time of requests. However notice though, since we request high consistency, every post request should still be directed to the leader, wherever it is located.

Reference

https://golang.org/pkg/net/rpc/#Call

https://golang.org/doc/articles/wiki/

https://github.com/golang/go/issues/13395