Overall design

In Lab 2, I extend and reimplement the web application of an online book store built in lab2. We have a multi-tier design, which consists of a front end and back end, each of the component provides different micro-services. The front end handled all the request by doing caching, load balancing and message forwarding. The back end consists of two parts, a catalog server handles all stock query and database update where the order server records and validate only buy request. I consider one catalog and one order server as one back end cluster and have 2 replicated clusters for fault tolerance and higher workload capacity. I used java Spark as micro-services framework and use dock container for easier deployment.

Caching

In this lab, I use 2 layers caching. All the caching allows concurrently read operation and in order update operation.

The first layer cache is on front-end server with 2 different caches. For the first one, all the static book information will be cached here since none of them will be changed by the user. So query request comes in, it will get information such as title and topics from here. As for volatile information such as stock, will be stored in the second cache which has weak consistency. When there is no information in the cache, the front end server will forward query request to catalog server and put the response information in the stock, and when there is a buy request updated the database, the order server will report it to the frontend server and the related stock information in the cache will be removed. Since we have a lot more query request than buy request, the information in the cache are accurate most of the time.

The second layer cache in on catalog server side, since all static book information will be handled by the front end cache, it only store stock information and has strong consistency with the database. It will be initialized when the server started and synchronized with the database after the DB is synchronized. Each time there is an update to database, it will block all the read request until the database update finished, the cache will be update first then all other query request or update request will be served. By doing so, all the query information from cache are accurate, so that we reduce the overhead to read from disk.

Load Balancing and Fault Detection

The system uses round robin load balancing approach. For each request that needs to be forward, the frontend server will randomly choose a cluster that are available. The available cluster id will be store in a special class which allows O(1) time removing crashed cluster id and randomly selecting an available cluster id. The selected cluster id will be used to choose corresponding server in different cluster, and the current algorithm and data structure already support load balancing for more replicas.

The frontend server will use heartbeat mechanism to detect failure server or the back online of the server. It will contact all the server every 5 seconds and update available server list accordingly. Beside of heartbeat, when no response from the server for a request, it consider that server crashed and resend the request to other server selected by the randomized round robin algorithm.

Disaster Recovery

When a catalog server restart from crash, it will query the latest data from another catalog server and ask it to lock the database until it finished the synchronization. It will passively wait the front-end server's heartbeat to reestablish contact and start to receive requests.

Trade-offs

In order to have consensus amount all database replicas, we use lock to prevent several buy requests to be processed at the same time.

In order to achieve strong consistence on catalog server side, we allow only catalog server to update the database rather than order server.

Possible Improvement

Currently each catalog server stores all data, in the future we can have different catalog responsible for different book request for even better load balancing.

Currently the system uses write to N read from 1 mechanism for consistence, and it is handled by one front-end, we can have move this responsibility to order server and have replication for better performance.