System Design

1.0. How to run the code

1. Ssh into elnux1 and login and clone repository in home diretory. Enter into lab-3-lab-3-gupta-kapadia
cd ~
git clone https://github.com/ds-umass/lab-3-lab-3-gupta-kapadia.git
cd lab-3-lab-3-gupta-kapadia
2. Change permissions of starter script: chmod +x src/starter.sh
3. Execute starter script: ./src/starter.sh
4. Enter password when prompted
5. To view logs in the same folder there are files- front_end.log, order_server.log and catalog_server.log
6. To view the output of the servers it is saved in each of the folders inside src/

1.1 Design Paradigms and architecture

We have divided the code into 4 different modules to make it more flexible.

● Config module : config3.txt : It consists of all the common constant values that will be used throughout the code-base. Since it is at one single place, any changes to be made can be done at one single place, instead of hard-coding and performing changes inside the main code. This will make the main code base more executable. We have configurable attributes for each server like:. order_ip1, order_ip2, order_port, catalog_ip1,catalog_ip2, catalog_port, frontend_ip, frontend_port.

●Catalog module:

1. Database_and_table_creation.py: It creates a database "Books_1" and inserts the records with their costs and quantities as specified.
2. Db_logger.py: This file is a sub module that provides the functionality to log write queries in case of a server failure. The queries are stored in db_logcatalog.csv. Each database replica has its own csv file.
3. Resync.py: This file contains code that resynchronises faulty database replica in case of failure
4. Catalog_server2.py: It implements the backend microservice, which maintains the catalog of books, ie. their title, quantity, cost and updates items by their item_id which is unique for all items. It registers the blueprint catalog_routes.py
5. configparse.py:This file parses the config file and command line args.
6. Catalog_routes.py: It is a blueprint of the catalog server which implements the following rest API's: All these APIs support both GET and POST HTTP requests. In the GET request we expect to get the request related information in the URL, while in the POST request, the request related information is sent in the json format.
   1. /list: lists all items in the database with all their attributes
   2. /lookup/:returns information about a particular item(number of items in stock and cost)
   3. /get_quanitity/:return quantity of an item specified by its id
   4. /update_c//:updates cost of an item specified by its id
   5. /update_q//:increases quantity of an item specified by its id, by amount
   6. /search/: return items of a certain topic Each of these REST API's return data in json format.
   .

● Order module:

1. Order_Server.py2: This is a backend microservice implemented to buy items. It registers order_routes blueprint.
2. Order_routes.py: It contains the following rest api's. All these APIs support both GET and POST HTTP requests. In the GET request we expect to get the request related information in the URL, while in the POST request, the request related information is sent in the json format.
   1. /buy/:Buys item specified by its id. Item's quantity is decremented by 1. We check if the quantity of the item is greater than zero before allowing the buy operation, otherwise we return an error message. This is implemented by issuing a get_quanity request to the catalog server and issuing an update _q request if the buy operation is valid, to decrement the quantity by 1.

● Frontend module:

1. frontend_Server2.py: This microservice implements the front-end part of the website which forwards all requests to the catalog server or order server after performing processing.
2. front_end_routes.py:This is the blueprint of front end server. It contains all the endpoints to the apis in catalog and order server.It also contains the invalidate cache endpoint, which implements a rest api to delete an entry of a item in cache when it is updated at the backend.(/invalidate/<item_id>)
3. Cache.py: This is a decorator functionality on top of all the read endpoints which implements a cache which stores item information in a dictionary by its id and pickles the dictionary object. It executes the endpoint function only if there is a cache miss.

● Starter.sh: The starter script installs required dependencies, creates database connection, table, inserts records, ssh's into remote edlab machines and executes all 3 servers in the background. It then executes the client test script based on the argument provided.

1.2. How it works?

The frontend service accepts all client requests and forwards them to the catalog or order server as required. Ideally the front end forwards the read requests to different server instances based on round robin, unless a server has failed. In that case it forwards it to the remaining server that is alive. If there is a cache hit, the requests does not go to the database. The write requests are forwarded to both the instances due to consistency. The catalog server handles lookup,list, update cost/quantity,search requests. The order server handles the buy requests by forwarding get_quantity and update_quantity requests to the catalog server. The order server performs some preprocessing and reroutes the final update to both the replicas of the catalog server. The cache entry is invalidated for a write request of a particular item before the execution of the database update. The logs are logged simultaneously while the catalog and order heartbeat servers at the backend sends messages to the front end via socket connection on a different port to update their status. If the heartbeats are not received then the server is said to have crashed and the requests are rerouted accordingly.

1.3. Architectural considerations and improvements:

Cache:We implemented in memory cache instead of REST calls since it reduces overhead of extra requests over the network. Setting up another server would also increase the number of points of failure, so we decided to keep an in memory cache at the front end. While implementing invalidation of cache entries during write requests, we created an api endpoint on the front end server which is called by the backend during write requests. As an **improvement** we could implement distributed caching using memcached implementation. As another **improvement** we would like to implement LRU cache and limit cache storage to reduce memory constraints

Cache consistency:Cache consistency is implemented by server push technique, which entails the backend servers to issue an invalidate request to cache in case of write requests. This leads to slight overhead in write requests, but ensures consistencies in cache. Upon receiving an update, front end server sends this update to the back-end server which calls the cache invalidate messages to caches to the front-end servers. Upon receiving of the invalidate message, the corresponding item from the cache is removed. Subsequent request causes a cache miss and the item is brought back into the cache.

Push based approach provides tight consistency because after every update, stale data from cache is removed. Cache consistency could be a problem in case of pull based approach. If the update and client request both come during the same poll period then client would be receiving the stale data. Hence, the poll period in pull based approach must be set properly to reduce this problem of cache consistency.

Load balancing: We have implemented round robin load balancing as compared to other load balancing techniques as it is simple and fair to all servers that are assumed to have equal capacities. In case a server crashes, we have rerouted requests automatically to the server which is up, so as to maintain a consistent state. Since there are only 2 server replicas, each server is allocated every alternate request, as long as the server is up. If a server is down , the request is re routed to the other server

that is up. This is repeated in a while loop to keep track . As an **improvement** we can use least connection load balancing which takes current server load into account and feeds new requests to least loaded server.

Load balancing in case of failure: In case of a failure the requests intended for the dead server are rerouted to the other server that is alive. This is assuming only one server will crash at a time. The rerouting is done at frontend for all requests except buy, which is done at the order servers.

Database synchronisation: Since we have two replicas for the database, we have to keep them consistent. We therefore route each write request to both the catalog server replicas from the front end. Another way to do it would be to reroute a request from one catalog server to another, but we felt that it would make a simpler design to have the catalog servers only route requests to their own replica databases.

Fault tolerance: We assume that there is at most one front end server failure at any time. The client should be agnostic of server fails. In our implementation, we allow a timeout(1s) period for each client request. If a server fails to answer in that interval, it throws an exception and then that client is reassigned to the other server replica .This is done because of the assumption that at most one server fails at any time. So if the current server fails, then according to our assumption all other servers are working properly hence, we assign that client to the other server.
Currently we detect failures using heartbeat messages from backend to front end , to ensure the backend replica hasn't crashed. We send messages to the catalog servers using a socket connection.
We allow a timeout(1s) period for each client request. If a server fails to answer in that interval, it throws an exception and then that client is reassigned to another server. The server status is stored in boolean arrays "is_catalog_server_active" and "is_order_server_active". The server information present is these arrays is updated every time the servers are polled by the front end heartbeat socket. This implementation also takes care of the fault recovery. Whenever a failed server becomes alive, its status is changed to 'alive' (1). The actual clients information for that server present in server fault tolerance info is then used to allot the clients to that server.
This isn't a very accurate way because a process being unresponsive does not mean that it has crashed. Choosing a timeout is hard, short timeouts can exacerbate the problem of inaccuracy, and long timeouts can make the system wait unnecessarily.

Restart server and resynchronise the databases: Before restarting the server we resynchronise the database replicas to avoid any inconsistencies. We execute the pending requests on the faulty replica from the write log of the accurate one.

Multi threading: To ensure progress, synchronization and prevent deadlocks - proper locking mechanism has been used in every piece of code - cache update, server status update during fault tolerance and recovery etc. Incorporating threading constructs to work around the reader-writer problem - whenever the back-end server updates its shared db file resource, it acquires an exclusive write lock. Further, for any get requests, it acquires a read-lock to access the updated information from the db and provide its consistent data to the client. The latter ensures no deadlock as well. Each Rest request to each server kind is maintained on a separate thread. Further, cache pull-consistency and sending heartbeat messages etc. are done in separate threads as well using the correct locking constructs. On the client side, 2 separate threads are run .Since SQLite 3.5.0, released in 2007, a single SQLite connection can be used from multiple threads simultaneously. SQLite will internally manage locks to avoid any data corruption. Further, read-write locks using the prwlock library has been made use of to support synchronization constructs. The respective end-points are defined in the "get" (read) and "post" (post) methods. Each new request to the server is handled in a different thread.

Exception Handling: In terms of correctness that given the right kind of inputs, our system won't have any logical or synchronization issues – we can guarantee that, since in cases when the database connection fails, we can send failure status as response and roll back the transaction. But handling exceptions in case of the front end server failure in the network, or taking care of a thread exception when it is still holding a lock etc. needs to be properly handled as it is not tested entirely. However, we've tried to incorporate exception handling constructs using the "try-except-finally" blocks and take care of such issues in the code. For example, locks are acquired in a try block and released in the finally block if an exception is caught midway .However, front end server is still the bottleneck for failure and performance

Logging: In every server all the requests, responses and function calls are getting logged. In our code we used simple text files to maintain the catalog and order log. Each server (Catalog and Order) only log the requests it received and that were executed on database replica. At the beginning of each run, the Catalog log include one or more entries recording the initial state of the catalog database (e.g. how many books were intially inserted), while the Order log is empty. The actual data of the databases is kept in the sqlite database.

EXTENTIONS:

Since the front end server can be a single point of failure we would like to extend the architecture in two ways:

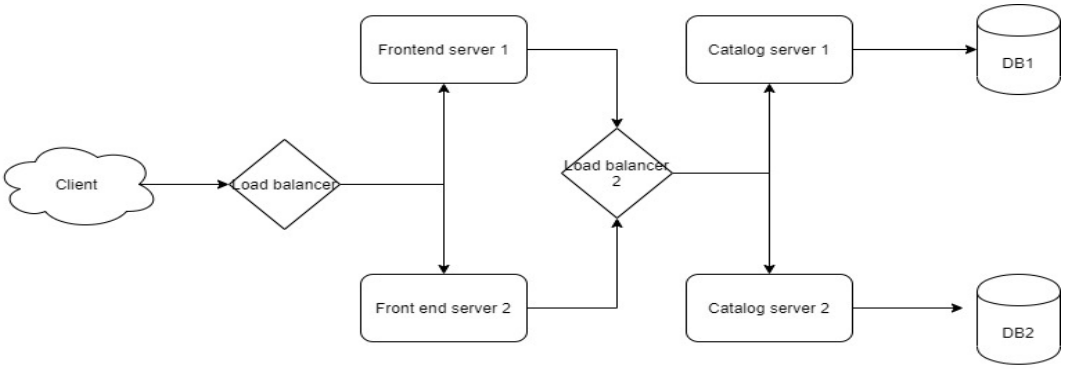
1.Add a extra load balancer in front of and after the front end and replicate the front-end server to reduce latency and speed up the response times and make the architecture more scalable.

2. Experiment with different load balancing techniques to see of they improve performance like :

-Least Connection Method — This method directs traffic to the server with the fewest active connections. This approach is quite useful when there are a large number of persistent client connections which are unevenly distributed between the servers.

-Least Response Time Method — This algorithm directs traffic to the server with the fewest active connections and the lowest average response time.

We propose the following changes to the architecture:



Text