Design Document CS677 Lab 3:

Pygmy.com employs a two tier web design. The front end consists of a frontEndServer which the client interacts exclusively with. The back end consists of an orderServer and a catalogServer with which the frontEndServer interacts with depending on the service required. Each component interacts with each other solely using REST API calls.

The system employs replication at the backend with each of the orderServer and CatalogServer having 2 replicas. These replicas maintain consistency in terms of their respective databases and are also fault tolerant up to one failure in each replication.

The front end also employs caching to improve read latency. A cache entry is invalidated when a write changes it value.

How to Run the System:

The system is built in Python 3. The system consists of 4 modules - client.py, frontEndServer.py, orderServer.py and catalogServer.py. In order to run the system -

- Install the following libraries - Flask, Requests, Flask-Caching.
- Specify IP and port of each server in the file - 'config.json' on all the machines that you wish to run the distributed system on. Map the server with its IP and port number in the following way - '"serverName" : "IP:port".
- Start all servers with the command 'python3 *serverName.py'* in any order making sure IP number specified in the config file matches the machine in which the server is started.
- Start the client using the command 'python3 client.py'.
- Follow the interface in the client window to perform actions.

Front End:

The client has the choice to perform one of three options on Pygmy.com. Search a topic for titles, lookup a book using its item number, buy the book he chooses using its item number. All these requests are routed to the frontEndServer. The frontEndServer then chooses to route it further to the backend to either the catalogServer(in case of lookups or search requests) or orderServer(in case of buy requests).

Modules:

- client.py: Consists of the interface between customer and store. Only consists of a main method.
- frontEndServer.py: All requests from client route through this.
  Methods:
  1. Search(HTTP GET request) - routes the topic chosen by the client to the catalogServer.
  2. Lookup(HTTP GET request) - routes the itemNumber chosen by the client to lookup to the catalogServer.

3. Buy(HTTP GET request) - routes the itemNumbe chosen by client to the orderServer.
4. invalidate(HTTP DELETE request) - Invalidates a cache entry based on the key received from the catalog server.
5. heartbeat() - Runs as a separate thread in the frontEndServer runs as a separate thread when the frontEndServer to constantly monitors whether all the backend servers are active. On change of status it informs all other backend servers of the change.

Back End:

The orderServer and catalogServer make up the back end. On startup a syncing phase ensures that both servers have the same state. This syncing is also used in the case of recovery when a server is restarted after failure.

Modules:

- catalogServer.py: Any changes or lookups to the catalog are routed to the catalogServer.
  Methods
  1. Search - returns a json object to the frontEndServer consisting of all titles under the topic of search query along with their item numbers.
  2. Lookup - returns a json object to the frontEndServer containing title, topic, cost, stock and item number
  3. Buy - returns a json object to the orderServer consisting of title purchased, its cost and topic it comes under. In case of no stock, a message is sent to the orderServer conveying this. The catalogServer also logs all purchases made into a csv file called ‘orders.csv’ and updates the stock in the ‘catalog.csv’ file.
  4. Restock(HTTP POST request) - returns a json object to convey whether stock was restocked or not. It runs in an infinite loop in a separate thread and scans items to restock periodically. In case a title is out of stock, a lock is acquired to update the ‘catalog.csv’ with new stock.
  5. update(HTTP PUT request) - This method is invoked by the other catalogServer replica to ensure consistency when the other replica makes a write to the catalog.
  6. heartbeat(HTTP PUT request) - The frontEndServer calls this method to convey state information of all backend servers.
  7. sync(HTTP GET request) - This method is invoked by the other replica on startup to sync state.

- orderServer.py: All buy requests are routed through the orderServer.
  Methods:

1. Buy(HTTP POST request) - routes a buy request from the frontEndServer to the catalogServer by first checking if there is enough stock for the title being purchased using a lookup request.
2. update(HTTP PUT request) - This method is invoked by the other orderServer replica to ensure consistency when the other replica makes a write to the catalog.
3. heartbeat(HTTP PUT request) - The frontEndServer calls this method to convey state information of all backend servers.
4. sync(HTTP GET request) - This method is invoked by the other replica on startup to sync state.

Design Decisions:

- All servers in the distributed system serve concurrent client requests using implicit threading offered by Flask's 'threaded' option.
- Atomicity of the any writes to catalog file during a buy is ensured by acquiring a lock when the catalog server executes a buy and is released when updation of the catalog file is complete. Thus two buy requests will only be served in a sequential manner even though requests can be made parallely.
- We have chosen to allow concurrent lookups even if a parallel buy is in progress. This is because any subsequent buy by the same client also performs a lookup before response. Thus, in case stock has gone down to 0, buy will fail and client will receive an 'out of stock' response. We foresee that a case where a successful lookup followed by a failed buy would be a rare occurrence and even such a case has been handled.
- The catalogServer also performs the action of restocking items that have depleted. This is done in a separate thread which scans through various titles periodically and replenishes it when it encounters no stock through a REST call. This is done atomically by acquiring a lock that enables the catalogServer to access the catalog.csv file.
- Current maximum stock is set to 50.

Caching and cache consistency:

- Flask-Caching has been used to perform query caching in the frontEndServer.
- A timeout of 100 seconds is used to perform LRU caching. Any entry older than 100s is invalidated.
- Cache consistency is employed upon any write that happens in the catalog by an invalidate request sent between the catalogServer and frontEndServer. This invalidates any entry in the cache with the key being the itemNumber of the book.

Replication:

- The orderServer and catalogServer have two replicas each.
- On startup, the replica that is spawned last requests the first replica to transfer its state and thus synchronizing the two on startup (the catalog file in case of the catalogServer and the orders record in cause of the orderServer).
- This same mechanism is used in case of a failure and restart of a particular server.

- Consistency is maintained between each replica upon any write made to the database by first ensuring that the other replica has recorded the change before making the change locally. In the case that the other replica fails to respond before timeout occurs, the operation is aborted and client receives a response confirming a failed operation.

Fault Tolerance:

- The catalogServer and frontEndServer are both single fault tolerant. Which means that in case one replica goes down the system still functions normally.
- On restart of the failed server, it synchronizes with the other replica and resumes normal functionality.
- In the case of a failure, all requests are routed to the server that is still active so that no requests are lost.

Dockerization:

- The application has also been dockerized.
- We have created docker files for each of the servers(frontEnd, catalog and order)
- The instructions to build images from the docker files can be found in the readme in the readme.

Possible improvements:

- The use of locks in the system for accessing catalog is a possible bottleneck in the system and a lockless implementation could improve latency for a bigger user base. In which case intermediate write ahead logs or buffers could be used to maintain atomicity of the database.
- We have made use of simple csv files for logging and cataloging. This could be replaced with a more production ready dB such as MongoDB which will also enable better user analytics on top of the website.
- The use of locks is also a potential source of a deadlock situation since two concurrent writes could cause both servers to wait for a response to confirm that it has updates its server before it has written to its own. This has currently been handled using timeouts and informing the client of the failed request. This has been chosen to ensure consistency between databases over availability.
- Alternatively, a retry mechanism could be applied where a ceiling value is set to the number of times a server can attempt to update the value of the database. This would ensure higher availability.