Lab 2 – Pygmy, The Book Store

Submission by:

1. Adarsh Kolya - akolya@umass.edu (Spire ID - 33018261)
2. Vignesh Radhakrishna - vradhakrishn@umass.edu (Spire ID - 32577580)
3. Brinda Murulidhara - bmurulidhara@umass.edu (Spire ID - 32578418)

Contents

1. Introduction

The objective of this project is to implement a multi-tier web application. The web application is a simple version of a books store wherein users can search for the availability of books by topic or using the unique identifier assigned to every book. Users can then buy books of their choice depending on the availability. REST APIs are exposed to perform basic CRUD operations as part of the backend and frontend microservices. Flask, a light-weight web framework is used to create the distributed web application. SQLite is used as the backend database.

1.1 Goals

1. Build the backend and frontend components of a two-tier web application wherein microservices are used at each tier.

The backend consists of the following:

a. A catalog server to maintain the catalog of books available. Books can be searched by topic or a specific book can be looked up using its unique identifier.
b. An order server to handle requests for buying books. It stores all the purchase orders.

The frontend consists of the below microservice:

a. Front end server that accepts search, lookup book and buy requests from clients and hits the relevant APIs of the backend servers to service the client request.

2. Expose the search, lookup, and buy functionalities as REST APIs in each microservice, wherein the client hits the REST endpoint, and the server services the client request.

3. Ensure concurrent requests from clients are handled without the system being in an inconsistent or erroneous state.

4. Store data in a persistent manner and ensure that the system can be restored to an earlier consistent state in case of failures.

5. Run the web application on multiple systems in a distributed fashion.

2. Components and Interfaces

The front-end server implements the following interfaces:

2.1 search (topic)

This is an interface provided to the user (client) to search for books in the store by topic. A title and a unique identifier (itemNum) are returned for each match. We design this using REST client/server architecture, and a high-level design is as below:

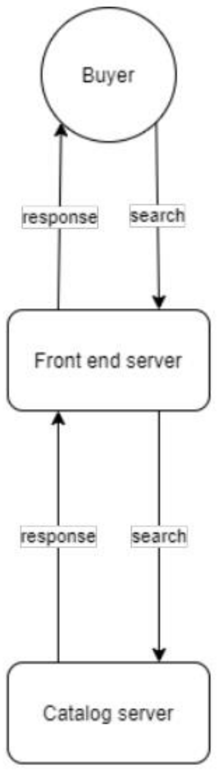


Figure 1 - Implementation details of the search interface

2.2 lookup (itemNum)

This is an interface provided to the buyer to look up the details of a particular book – availability of stock and price. The front-end server sends an API request to the catalog server in order to get details of a specific item in the store and forwards that response to the buyer.

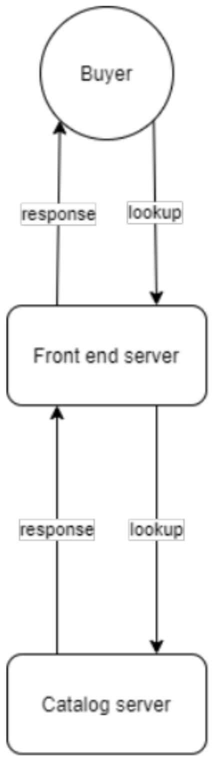


Figure 2 - Lookup interface

2.3 buy (itemNum)

This is an interface that has been provided to the buyer. Buyers can use this to buy a book from the store. The front-end forwards this request to the order server using the buy API exposed by the order server. The order server checks the availability of the book by hitting the API exposed by the catalog server to get details of a book (price, stock). If there is availability, the order server hits the catalog server API to decrement the stock count. The catalog server then returns a success/failure response which is forwarded to the buyer through the order server and front-end server. Below is a high-level design of how this works.

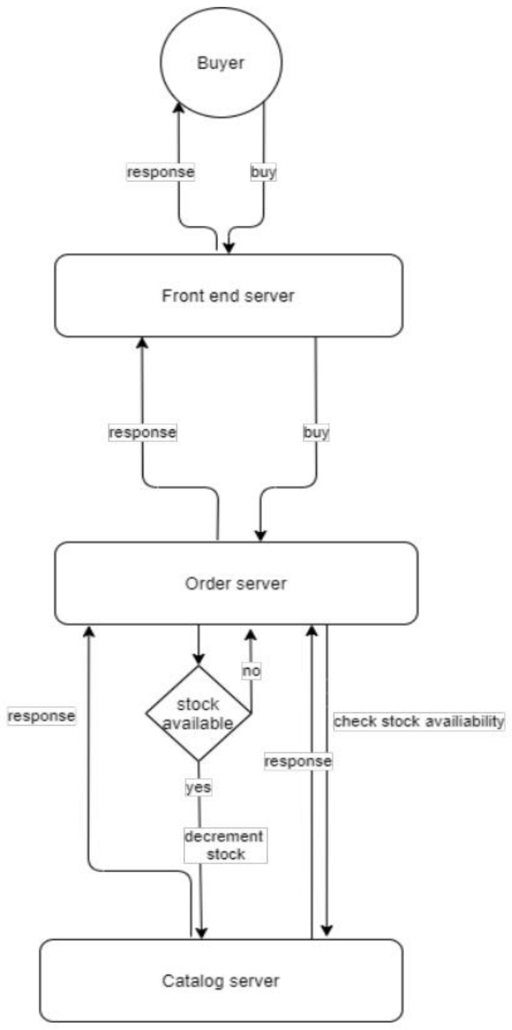


Figure 3 - Buy interface.

3. How to run the system?

Running the servers

Create a machines.txt file and store three lines for the machine addresses for catalog, order and frontend servers respectively. Install python3 on all machines that you are using to run the servers and clients.

```
bash run.sh machine.txt
```

For running the entire system on your localhost, use machines.txt.local.

```
bash run.sh machines.txt.local
```

CLI

Open a new shell and run the following commands

```
cd LAB-2-PYGMY-THE-BOOK-STORE
```

For linux based

```
source .venv/bin/activate
```

For windows based

```
.venv/scripts/activate.bat
```

```
 cd src/cli
```

Lookup

```
python main.py lookup <item number>
```

Example

```
python main.py lookup 1
```

Search

```
python main.py search --topic "<topic>"
```

Example

```
python main.py search --topic "distributed systems"
```

Buy

```
python main.py buy <item number>
```

Example

```
python main.py buy 1
```

3.1 Running test cases

3.1.1 Run testcases for Single server, single client usecase

```
cd Lab-2-Pygmy-The-book-store
chmod 777 test/SingleServerSingleClient.sh
bash test/SingleServerSingleClient.sh
```

3.1.2 Run testcases for Single server, multi client usecase

```
cd Lab-2-Pygmy-The-book-store
chmod 777 test/SingleServerMultiClients.sh
bash test/SingleServerMultiClients.sh
```

3.1.3 Run testcases for Multi server, multi client usecase

Create a file "machines.txt" with 3 lines.

First line has the IP of the server where catalog server is to be launched. Second line has the IP of the server where order server is to be launched. Third line has the IP of the server where frontend server is to be launched.

The machines.txt file should either contain all entries corresponding to localhost or ec2. A combination of the two is not supported currently.

Please ensure the below:

1) Test cases are being run from a 4th machine. No servers should run on this machine.

2) This should be a different machine with passwordless ssh enabled to the other 3 machines.

Example for contents of machines.txt:

```
ec2-35-175-129-185.compute-1.amazonaws.com
ec2-54-167-65-75.compute-1.amazonaws.com
ec2-54-85-5-151.compute-1.amazonaws.com
```

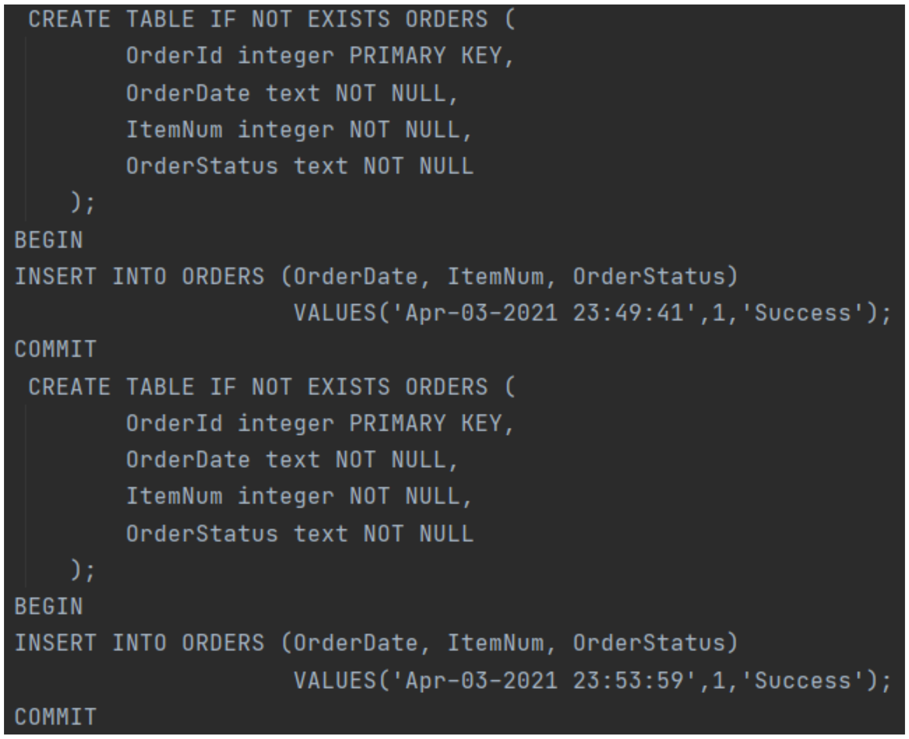
```
cd Lab-2-Pygmy-The-book-store
chmod 777 test/MultiServerMultiClients.sh
bash test/MultiServerMultiClients.sh machines.txt
```

3.3 Logging

Both the catalog server and order server log every SQL query that is executed in order to restore the database to a prior state in case of failure and loss of data. The queries are written to a file query_log.txt sequentially after every database call.

Order server

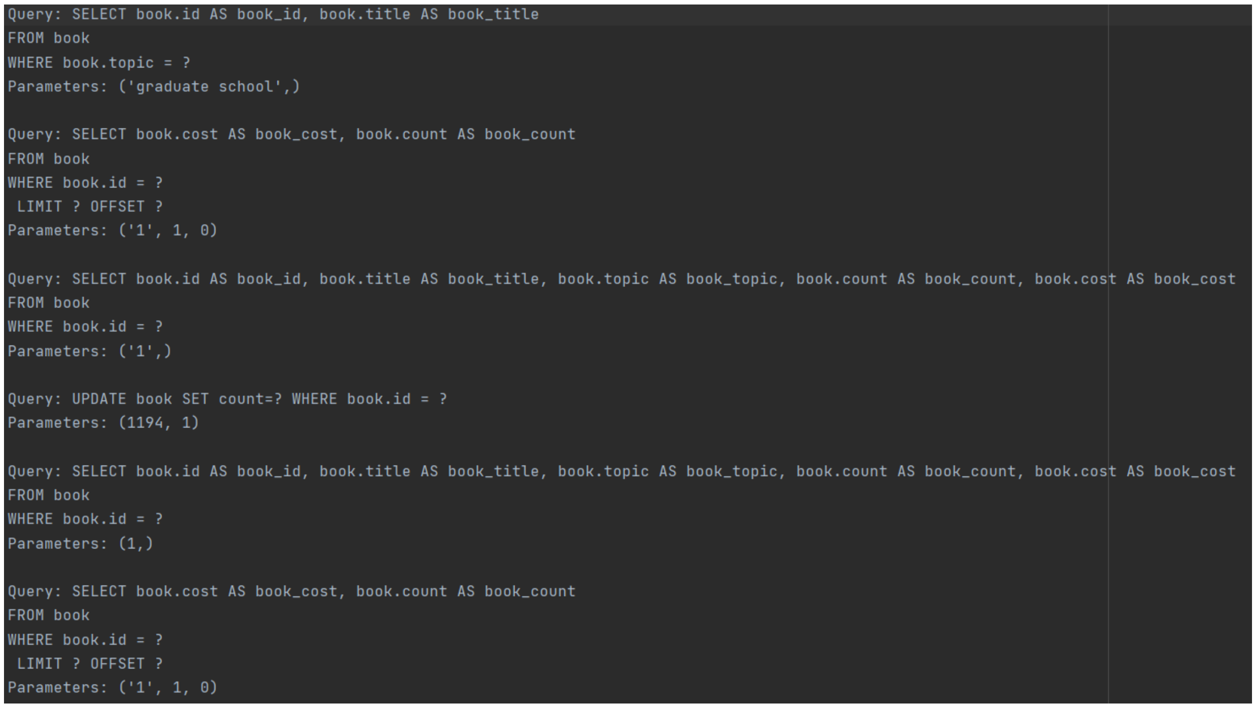
Below is a sample of the query log. The queries can be demarcated using the “BEGIN” and “COMMIT” keywords and “;” indicating the end of a query. Queries are logged after every SQL statement is executed.



```
 CREATE TABLE IF NOT EXISTS ORDERS (
        OrderId integer PRIMARY KEY,
        OrderDate text NOT NULL,
        ItemNum integer NOT NULL,
        OrderStatus text NOT NULL
    );
BEGIN
INSERT INTO ORDERS (OrderDate, ItemNum, OrderStatus)
                    VALUES('Apr-03-2021 23:49:41',1,'Success');
COMMIT
 CREATE TABLE IF NOT EXISTS ORDERS (
        OrderId integer PRIMARY KEY,
        OrderDate text NOT NULL,
        ItemNum integer NOT NULL,
        OrderStatus text NOT NULL
    );
BEGIN
INSERT INTO ORDERS (OrderDate, ItemNum, OrderStatus)
                    VALUES('Apr-03-2021 23:53:59',1,'Success');
COMMIT
```

Catalog server

Below is a sample of the query log. The ‘Query’ key refers to an SQL query and ‘Parameters’ refers to the set of values to be substituted for the columns in the query.



```
Query: SELECT book.id AS book_id, book.title AS book_title
FROM book
WHERE book.topic = ?
Parameters: ('graduate school',)

Query: SELECT book.cost AS book_cost, book.count AS book_count
FROM book
WHERE book.id = ?
 LIMIT ? OFFSET ?
Parameters: ('1', 1, 0)

Query: SELECT book.id AS book_id, book.title AS book_title, book.topic AS book_topic, book.count AS book_count, book.cost AS book_cost
FROM book
WHERE book.id = ?
Parameters: ('1',)

Query: UPDATE book SET count=? WHERE book.id = ?
Parameters: (1194, 1)

Query: SELECT book.id AS book_id, book.title AS book_title, book.topic AS book_topic, book.count AS book_count, book.cost AS book_cost
FROM book
WHERE book.id = ?
Parameters: (1,)

Query: SELECT book.cost AS book_cost, book.count AS book_count
FROM book
WHERE book.id = ?
 LIMIT ? OFFSET ?
Parameters: ('1', 1, 0)
```

4. Areas of improvement

1) **Load balancing:** We could have multiple replica servers running the same set of microservices and a load balancer could divert the incoming requests based on the load of each server at that moment in time. This would avoid single servers becoming a bottleneck when the system scales.

2) **Caching:** The front-end server can implement a caching layer to cache books under a certain topic. Most users are likely to browse the catalog for various books and buy a few of them. As the title, the identifier for a book, and the topic of the book are not likely to change frequently, this data can be cached. The responses from the front end would then be much faster and it would also reduce the load on the catalog server.

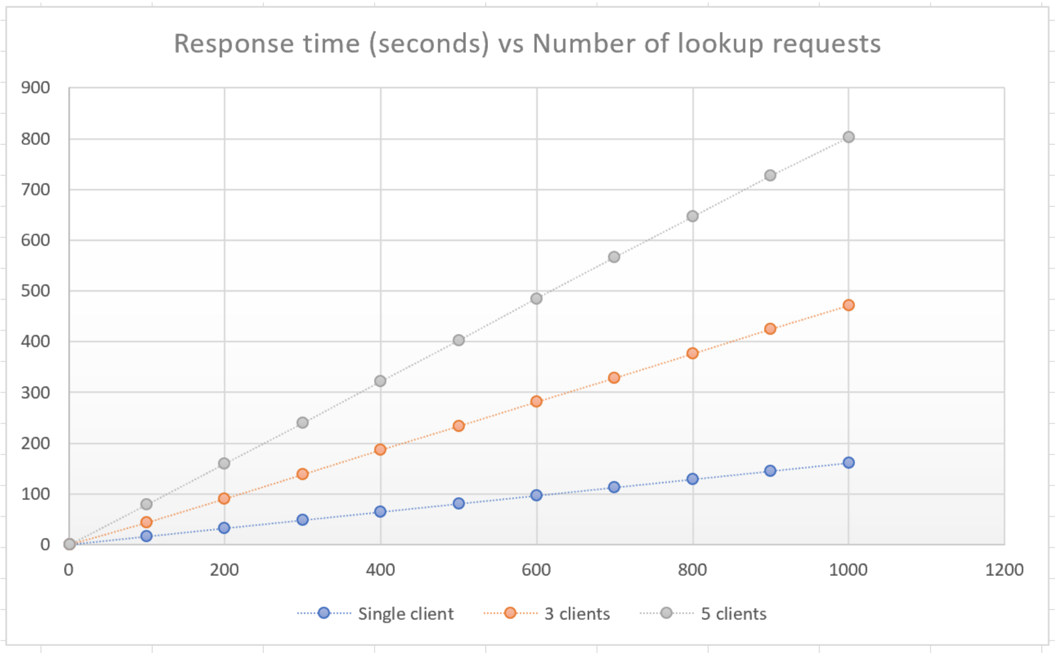
3) **Backup and recovery:** Backups of the databases must be taken regularly to avoid losing data permanently when there is a system failure.

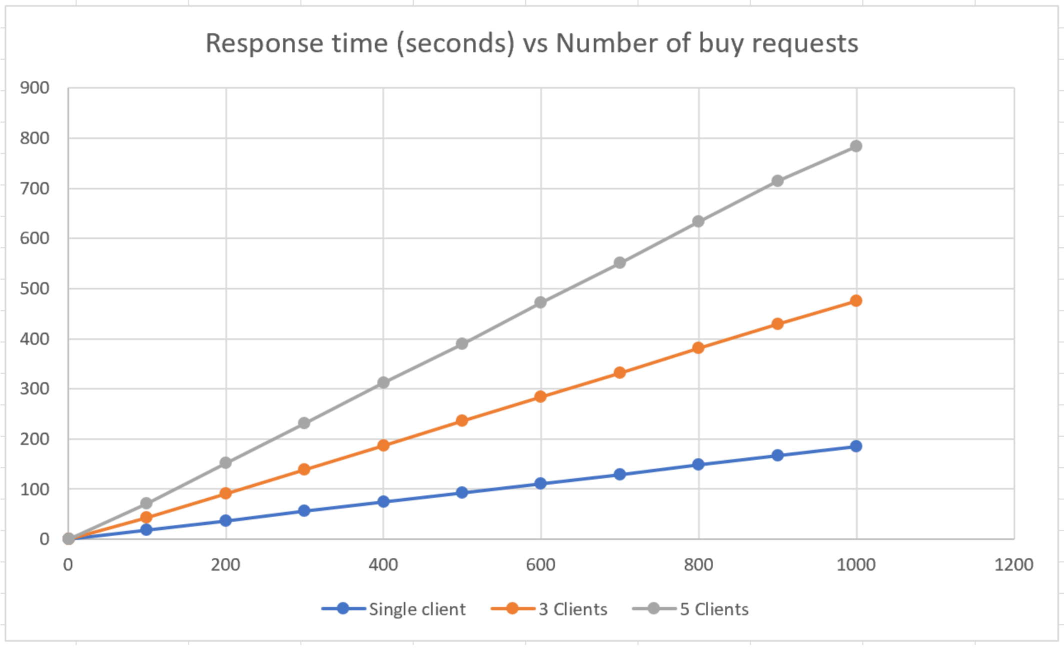
5. Design Trade-offs

1) We use Flask as it is lightweight. But Flask is not well known for handling concurrent requests and so as number of clients would increase, the system will become extremely slow resulting in a bad user experience.

6. Performance

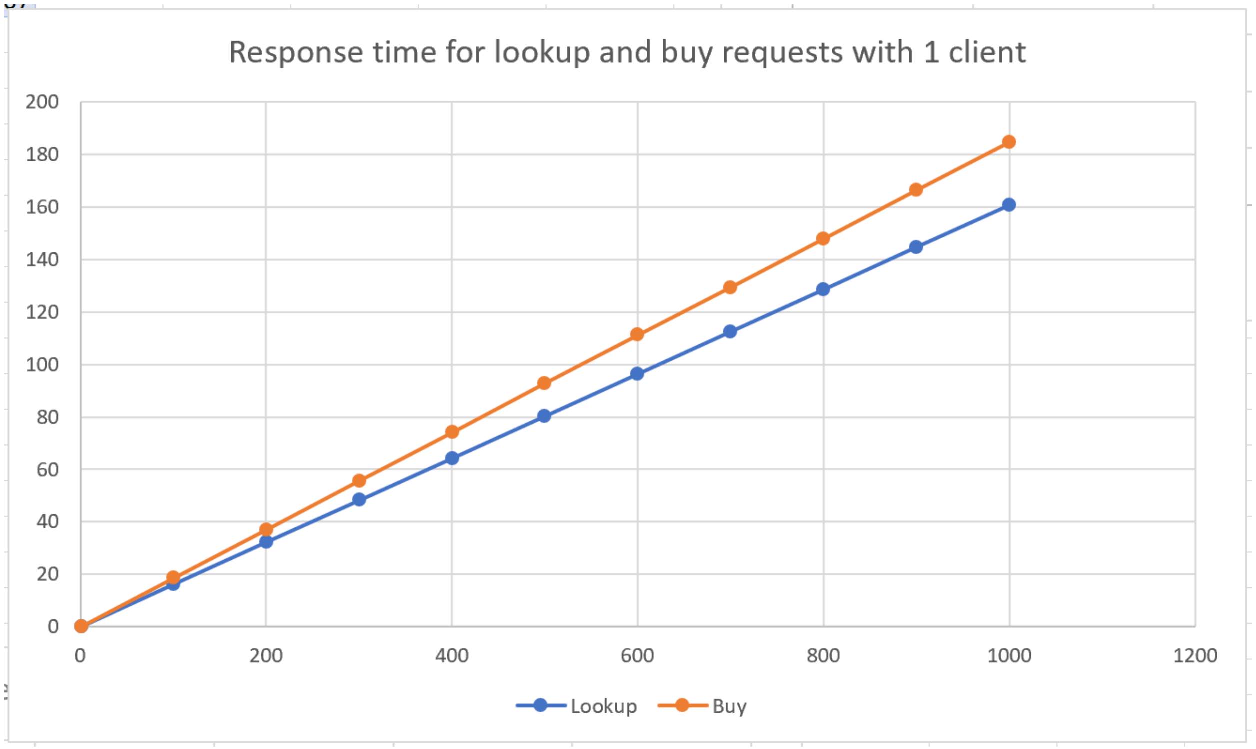
Below is an analysis of how the response time changes with the number of simultaneous clients.





It can be observed that the response time increases as the number of simultaneous clients increase.

Below is an analysis of how response time differs when multiple layers are involved. For lookup() requests only 2 layers and involved. For buy() requests all 3 layers are involved.



From this analysis, seems like the system does not show a lot of latency when multiple layers are involved.

We can conclude that when number of clients increase, the response time increases, but the involvement of multiple layers, did not significantly increase the response time.