CS667 Distributed Operating Systems

Spring 2019

Lab 2 Design Document

Team Name: STIMA

Name: Aggrey Muhebwa
Github Handle: amuhebwa
Student ID: 32055729

Name: Zeal Shah
Github Handle: zealshah95
Student ID: 31737150

Multi-tier Architecture Design

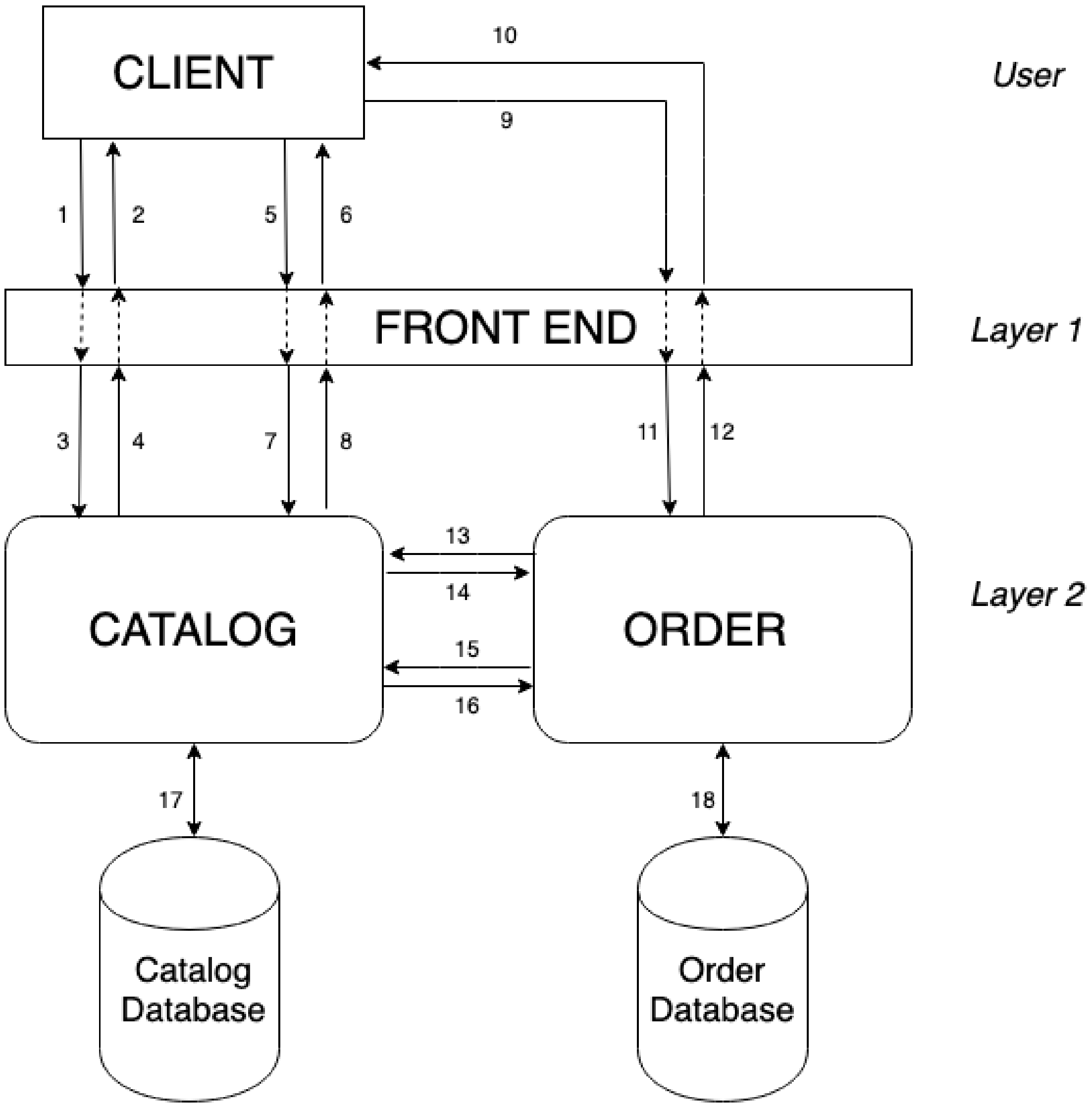

Figure 1: Two-tier Web Design

In layered architecture, a layer can only talk to the layer above it and to the layer below it. A similar layered architecture has been implemented for our web framework. As shown in

Figure 1, the architecture of our book store has 2 layers- layer 1 is the front end server and layer 2 consists of catalog server and order server. Layer 1 can only talk to the client and layer 2. Layer 2 can only talk to layer 1. Client and layer 2 cannot interact directly, the interaction has to take place only via layer 1. Each layer exposes a bunch of interfaces that can be used by relevant layers for communication purposes. Interface for each component is implemented as a HTTP REST interface. Front end, catalog and order servers are multi-threaded. They can handle multiple requests concurrently.

Client contacts the front end server for the purpose of exploring the contents of the book store and also for buying a book of interest. Front end server forwards the requests from client(s) to back end server(s) and forwards the response(s) received from back end server(s) to the client(s).

Catalog server maintains the catalog of the book store in a catalog database. Front end server contacts the catalog server for any search related queries sent in by the client like- books by topic, details of a specific book by book number. Based on the nature of query request, catalog server checks the catalog database, and sends out the relevant information to the front end server. Catalog server provides with update interface which can be used to update stock count and price of a book. Stock of every book is checked periodically using timer threads in the catalog server. Stock of a book is then updated if it's stock has gone empty.

Order server handles the transactions happening in the book store that is buying of a specific book. Front end server contacts the order server when a client sends in a buy request for a specific book number. On receipt of the buy request, the order server verifies with the catalog server that the requested item is still in stock. If the item is in stock then the order server sends an update request to the catalog server to decrement the count of the requested item and marks the purchase as successful.

Details regarding database design, HTTP REST interface, and operations are discussed in the subsequent sections.

Database Design

Two approaches are leveraged to store data

- The catalog database is implemented as an SQLite Database
- the Order database is implemented as a simple comma separate values file (csv) bound over pandas data-frame

Catalog Database

The catalog database is implemented as an SQLite database. When the Catalog server first starts, it creates a new database and subsequent restarts will just (1) check if the database does not exists, create it and (2). If the required tables do not exist, create them. The database contains a single table, 'books' with five fields for the book number, the book title, the topic under which the book falls, the amount in stock, and cost of that book. When the database is initially created, the table is also create and pre-populated with 4 books in two categories. We can perform the standard database operations; Create, Read, Update and Delete (CRUD) operations on catalog database, expose via intermediate functions. As an added security measure, we open a connection at the start of the query and close it once the query completes. While this might cause performance overhead, it ensures that a single connection and cursor is create for a specific function when interacting with the database.

Order Database

The order database is implemented as comma separated values (csv) file, with binding to pandas data frames. The csv file is create the first time the order server starts and for each transaction, the file is loaded, and the transaction logs recorded. The file contains four columns; The item number that the client requested to buy, the status of the transaction, that is whether it was successful or if it failed, the time stamp indicating when the order was transmitted and finally, the quantity of the item remaining in stock. Our design choice for using a csv file to log transaction vs using a database if based on the notion that as the system grows in size (our system will one day grow to the size of Amazon), the amount of transactions in a day become too many to be logged into the database without running out of storage space. On the other hand, if the logs are stored in files, these files can be archived after t hours, in-case the server is running out of space.

HTTP REST Interface Implementation

The code uses HTTP requests and responses for distributed communication. Different interfaces for catalog, order and front end servers have been implemented as HTTP REST interfaces. Arrows shown in Figure 1 represent the back and forth communication taking place between different entities. We refer to the arrow numbers shown in Figure 1, to discuss the REST interfaces and their responses. NOTE: All the request calls are HTTP REST API calls and all the responses are JSON objects. Only numbers (17) and (18) do not represent HTTP REST API calls. Catalog server and order server use function calls to query and update their respective databases which is shown by (17) and (18) in Figure 1.

Front End Server Interfaces

Front end server provides three different REST interfaces-

- *search_request(topic)*- shown using (1), it allows the client to query for books related to a specific topic. Response shown by (2), contains a list of related book titles and book numbers. In the background, front end server triggers a query to the catalog server by using *query_by_subject(topic)* API of catalog server shown using 3.
- *lookup_request(book number)*- shown using (5), it allows the client to look for details associated with a specific book. The response (6), contains details like cost and stock count of the book. On receiving the lookup request (5), the front end server triggers a query to the catalog server by using *query_by_item(book number)* API of catalog server which is shown by arrow (7) in Figure 1.
- *buy(book number)*- Client uses this API to place buy request for a specific book (9). Response (10) to the buy request contains a message associated with transaction's status- successful or failed. In the background, front end server forwards the buy request to the order server using order server's *buy_request(book number)* API (11).

Order Server Interfaces

Order server provides one REST interface-

- *buy_request(book number)*- shown by (11), allows the front end server to specify a book number that client wants to buy. Uses catalog server's *query_by_item(book number)*

API (13) to verify the stock of the requested item and then uses catalog server's *update(operation, change, book number)* to decrement the stock by 1. Specifically, order server makes *update(decrease_count, 1, book number)* (15) call to decrement the stock value by 1. Response to front end server contains the status of transaction(12).

Catalog Server Interfaces

Catalog server provides three different REST interfaces-

- *query_by_subject(topic)*- front end server uses this API to request for list of book titles associated with a specific topic (3). Catalog server checks the database and sends the list of related book titles and numbers to the front end server in response (4).
- *query_by_item(book number)*- front end server uses this API to request for details regarding a specific book using its book number (7). Catalog server checks the database and sends a response (8) containing the stock count and price of the specified book. Same request (13) and response (14) operations take place for order server using this API to verify stock of a specific book.
- *update(operation, change, book number)*- it supports three different operations
  - *decrease_count* operation is used to decrease the count of a specific book in the database by amount specified in *change*.
  - *update_cost* operation is used to change the cost of a specific book in the database to value specified in *change*.
  - *add_stock* operation is used to add new stock for a specific book. The new stock value is specified in the *change* variable.

How Does it Work?

Catalog server, order server, front end server and the client can be located on same or different machines. Client is programmed to use only the interfaces provided by the front end layer. Front end layer then uses interfaces provided by the back end layer for different operations.

Client uses *search_request(topic)* interface of front end server to send a request for books related to one of the two topics- distributed systems or graduate school. On receiving this request, the front end server makes *query_by_subject(topic)* call to the catalog server. When the catalog server receives this request, it looks for all the books related to the requested topic in the catalog database. If any relevant books are present, it returns a JSON object containing list of books with their title and number to the front end server. The front end server then forwards the JSON object to the client as a response.

Once the client receives a list of books to choose from, the client selects one of the books at random and tries to lookup for details of that book using front end server's *lookup_request(book number)* interface. In the background, this request is forwarded to the catalog server through *query_by_item(book number)*. On receipt of the lookup request, the catalog server queries the catalog database and sends the details like cost and stock count of the book to the front end server. These details are then forwarded to the client.

The client checks if the book is available in stock and only then places a buy request to the front end server using *buy(book number)*. Front end then forwards the buy request to order server. As soon as it receives the buy request, the order server verifies the availability of the book in stock using catalog server's *query_by_item(book number)* interface. If the book is available in stock, the order server makes *update(decrease_count, 1, book number)* call to the catalog server which basically decrements the stock count of that specific book by 1. Catalog server then updates book's stock count in the database. Order server then marks the transaction as successful and sends the message to the front end server which then gets sent to the client. In case the order server sees that the book stock is empty, it sends a message that the transaction cannot be completed as the book is out of stock. During this complete process, order server logs the transaction details in the order database along with their timestamp and status.

Timer threads have been used in catalog server that periodically check if any book has run out of stock. New stock is added if any book is out of stock using the update interface. So, periodical stock updates are done in the catalog server itself.

In place of client.py, if the user chooses to run sequential_requests_times.py, it will send 1000 sequential search, lookup, and buy requests. Also, concurrent_requests.py can be used to send concurrent requests. The details of sequential and concurrent requests have been discussed in the performance evaluation document.

Exception Handling

In order to ensure correct operation of all the entities, we have resorted to basic exception handling. Some basic exceptional scenarios can be-

- Client sends a search request for topic not in the database- In such a case, the catalog server after querying the database, will send a response stating that there are no books available for the given topic.
- Client sends a lookup request for a wrong book number- In this case, the catalog server will behave as mentioned above and will send a response stating that there is no such book in the store.
- Client sends a buy request even if the stock was empty- This has been implemented in the 1000 sequential requests code where the client will send 1000 sequential buy requests regardless of the stock count. In such a scenario, the order server's verification step will come to the rescue. If the order server finds out that the stock is empty then it sends a message stating book is no more available in the store.
- Client sees the book is in stock and tried to buy it, but by the time the buy request reaches the order server, the stock goes empty- Again the order server stock verification will prove to be helpful here. When the order server finds out that the stock is empty, it sends a message stating book is no more available in the store.

Program output snapshots and the test document can be referred to for more details regarding the system behavior and outputs in such scenarios.

Design Trade-offs

- GET is used to retrieve data and to make updates.
- For evaluation of concurrent requests handling, we generate a bunch of threads, let them execute for some specific amount of time and then kill all of them together. The end to end response time measured is the difference between the time when a bunch of threads were created and the time when all of them were killed. It doesn't take into account the time taken by every thread to finish execution.

- Two different types of databases are used- catalog database is SQL database and order database is a CSV file.
- Client can only buy 1 item at a time. Bulk purchase is not supported.
- Update interface contains 3 methods. Method names are used as variables in the update REST Call API.

Possible Improvements

- Following the standard REST specifications, implement use of POST instead of GET to make updates.
- Figure out an efficient and more accurate way to deal with threads for the purpose of sending concurrent requests. Also, figure out a better way to measure response times when multiple clients are making concurrent requests.
- Use just one kind of database preferably SQL for both catalog and order.
- Support bulk order purchase. At present, the catalog server supports bulk decrement of stock based on the number stated in *change* of update interface. But the order server will always send change = 1 in the update interface. So, whenever a customer places a buy request, it is assumed that the customer will buy just 1 item. A feature can be added where along with the book number, client can specify the quantity of books that it wants to purchase. Order server will then verify if enough items are available in stock to fulfill the order, if yes, it will use the change = quantity in the update interface.
- Instead of using 3 methods inside update interface, we can create 3 different REST APIs.
- Add more features to the databases for better analysis.
- Instead of making databases a part of the catalog and order servers, we can implement a separate data layer.
- We can combine the order and catalog servers to simplify the system and its operations.
- Utilize better databases in order to implement transactions.

Implementation/Running the Program

NOTE: We have hard-coded IP addresses of the EdLab machines in our servers' code instead of using a configuration file. In order to ensure correct operation of our system, please make sure that the servers are run on the exact same EdLab machines as specified in the steps given below.
Steps to run the setup on distributed machines:

- **Step 1:** Copy the lab-2-stima folder to *EdLab 1*.
- **Step 2:** Ensure that flask, pandas and matplotlib modules are installed on *EdLab 1, EdLab 2, EdLab 3* and *EdLab 7*.
- **Step 3:** All the files should be run with Python3 as *python3* {*file_name.py*}. The reason behind this instruction is that in EdLab, the modules in step 2 get installed under python 3 instead of python 2.7.

- **Step 4:** Start the servers in the following order- *catalog.py* MUST be started on *EdLab 1*, *order.py* MUST be started on *EdLab 2*, *front_end.py* MUST be started on *EdLab 3* and then all the client files(*client.py, sequential_requests_time.py, concurrent_requests.py*) MUST be run on *EdLab 7*.
- **Step 5:** To run functional tests for *front_end.py*, the file *test_frontend.py* MUST reside on the same server as *front_end.py*, in this case, *EdLab 3*.

Evaluation

Messages associated with all the requests, responses and transactions are displayed in the terminal window of every node. Sample outputs have been provided in the DOCS directory. Test document contains all the details regarding the tests conducted, expected results and actual results obtained. Performance evaluation document contains details and outputs of performance evaluation experiments like response times seen for sequential and concurrent requests.