CS677 Lab 2 Design Doc

Kamesh Balasubramanian, YenSung Chen

Overall System Design

The task was to design a book store selling exactly four books. The overall service follows a client-server architecture that consists of a front-end and a backend. The actual backend is split into two seperate microservices. So we have three microservices in total: One for the front-end and two for the backend. The front-end is the service that clients of the book store interact with to make their requests. The front-end then relays the requests to two backend services depending on the type of request from the client: A catalog server or an order server. All the services/components were built with Python's Flask framework.

Backend Details

As mentioned above, the backend consists of two seperate microservices. The catalog server and the order server.

Catalog Server

The catalog server is a Flask server that serves three kinds of requests through RESTFUL APIs: query-by-topic, query-by-itemno and update. The query-by-topic is implemented as the search(topic) method. This method is exposed as 'GET' endpoint. The front-end makes an http GET request to the method's endpoint and the method finds all the books with the requested topic and returns the relevant information. The query-by-itemno is implemented as the lookup(itemno) method and is similar but finds a unique book and returns that back to the front-end. The update(itemno) function is used to adjust the stock of a given book once it is succesfully bought by a client. It is exposed as an http 'PUT' endpoint. It reduces the stock of the book with the input item number by 1. Since Flask spawns a thread for each request, this can handle concurrent requests from multiple clients.

Implementation Details

The API of the catalog server is as follows:

- *search*(*topic*) (http GET): This method receives an input topic from the frontend and returns all books that match the topic.
- *lookup*(*itemNum*) (http GET): This method receives an input item number from the front end and returns the book with the item number.
- *update*(*itemNum*) (http PUT): This method receives an item number as a parameter and updates the stock of that item. The new value of the stock is part of the request body.

Order Server

This is also a Flask server. It only accepts a buy request exposed as an http GET endpoint. When the front-end makes the GET request to the endpoint corresponding to the buy method, the order server makes a query-by-itemno request to the catalog server to verfiy that the item is in stock. If it is in stock, it makes an http PUT request to the update endpoint of the catalog server to decrease the stock of the book by 1. It then sends a message back to the front end saying the buy was succesful. If the item was not in stock, it replies with a message saying that the buy failed because the item was out of stock.

Implementation Details

The API of the order server is as follows:

- *buy*(*itemNum*) (http GET): This method receives an item number from the frontend. It then makes a GET request to the *lookup*(*itemNum*) endpoint of the catalog server. If the item is in stock, it decreases the stock by 1 by making a PUT request to the endpoint of *update*(*itemNum*) of the catalog server and returns the book to the frontend. Otherwise, it returns a message saying that the buy failed.

Front-End details

The front-end is simply a service accessed through a command line interface. It is implemented as a Flask server that accepts two types of requests: A search request or a buy request. These are implemented as REST endpoints.

Search Requests

A client may make a search request to the front-end by making a http search request to the search endpoint in the frontend with either a book topic, or an item number for a specific book. If the request is made with a topic, the client will get a list of all the books available in the backend with the same topic. If a request is made with an item number, the front-end returns the client the book with that unique item number. The front-end does not have access to the books directly. On receiving a search request, the front-end then makes the correct http search/lookup request to the catalog server in the backend to retrieve the requested information.

Buy Requests

A client makes a buy request by making an http request to the buy endpoint in the front end with an item number. The frontend then makes a http request to the corresponding buy endpint in the order server in the backend to try and buy the book.

Implementation Details

The frontend API is as follows:

- *search*() (http GET): This method receives either a topic or an item number in its request body. Based on this input, it makes the corresponding call to *search*(*topic*) or *lookup*(*itemNum*) of the catalog server.
- *buy*() (http POST): This method receives an item number in its request body and makes the corresponding call to the *buy*(*itemNum*) method of the order server.

How the System Works

Any client makes requests (search or buy) to the front-end server by calling the respective http function to the respective endpoint. Depending on the request the frontend then makes an http request to either the catalog server (for searches) or the order server (for buys). The reponse is then relayed back to the client.

Our Client

Our client randomly chooses to make a buy or search request to the front-end. This is done in an infinite loop. After sufficiently many succesful buy requests, the stock of all the books become 0. After this stage, all subsequent buy requests fail.

Storing the inventory

The inventory of books are stored in a csv file. Each line of the csv file has data for a single book: the item number, name, stock, cost and topic. We use a csv file and not a lightweight database like sqlite since the inventory we use isn't very big. We need to have our own lookups, and updates because of this as we do not have an API to call. Also, the inventory is a shared resource which isn't thread/process safe.

Ensuring consistency of data across multiple clients

The SQlite database is threadsafe. Since Flask spawns a thread for each process, this would be a good choice for a database to store the inventory. However, our inventory isn't very big and the SQLite docs do not recommend using SQLite when there are multiple threads. Since we use a simple csv file, we use a bounded semaphore with bound 1 to avoid race conditions. We acquire the semaphore before every search, buy or update operation. Since, each flask server is one process that spawns multiple threads, this semaphore is shared and ensures serial access to the inventory (even though the requests from are concurrent).

Tradeoffs and Improvements

We could use SQLite instead of a csv file for the inventory (despite the recommendation not to use SQLite in a multithreaded application). Also, we use Flask in development mode served directly. However, this is not a good idea if we want to scale the service. However, considering the size of the service, we chose to deploy directly. If we wanted to scale the application to handle a much higher number of users, we would need to deploy Flask as a wsgi module on a wsgi server. We could also use something like postgreSQL that scales better and supports a higher level of concurrency than SQLite to handle large inventories and much larger number of clients. This would also eliminate the need for us to ensure data consistency as that is provided by the database.