LAB 2 DESIGN DOCUMENT

SHREYA BIRTHARE
SIMRAN MALIK

INTRODUCTION:
This lab is an extension of the first lab. The Online Toy Store System is designed to handle people shopping from the toy store's catalog with various options such as querying the availability and price of the toy or buying the toy. The system is built to manage multiple client connections simultaneously, providing real-time information on toy prices and stock levels. We employ a two-tier design for the Toy Store (a front-end tier and a back-end tier) using microservices at each tier. The front-end is implemented as a single microservice, while the back-end is implemented as two separate services: a catalog service and an order service. This document outlines the system's goals, proposed solution, detailed design, along with evaluation metrics.

GOALS AND OBJECTIVES:
The goal of this lab is to:

- Design distributed server applications using a multi-tier architecture and microservices.
- Design virtualized applications.
- Design interfaces for web applications.
- Learn how to implement a REST API server.
- Learn to measure the performance of a distributed application.
- Learn how to use Docker to containerize your micro-service and learn to manage an application consisting of multiple containers using Docker Compose.
- Learn how to test a distributed application.

DESIGN DETAILS:

Part 1

The task is to implement “Asterix and Tiered Microservices-Based Toy Store” and design distributed server applications using a multi-tier architecture and microservices.

The entire code is written using python.
The front-end is implemented as a single microservice, while the back-end is implemented as two separate services: a catalog service and an order service.
HTTP Rest APIs are used for communication.

Front end service

The front end service uses inbuilt BaseHTTPRequestHandler, ThreadingHTTPServer from http.server library in python to handle incoming requests concurrently using threads.
Thread per session model is used to process client requests.
It starts on the specified port: 12503
and host ip and listens for requests indefinitely. (ip of the set up/machine where the service will run- can be modified according to set up used for running the application).

FrontendHandler class handles all the request that come to the front end server.

do_GET method:
The do_GET method handles all the get requests that frontend service receives.
It first checks if the parsed path starts with "/products/", indicating it is a query request which will directly go to catalog. At the end of the parsed path, the name of product resides for which corresponding details like quantity and price are requested. This product name is extracted and communicated to catalog service using url:
http://{CATALOG_HOST}:{CATALOG_PORT}/{product_name}
The response from the catalog is stored in product_info variable. If the received status code is 200, that means query was successful and the received product details are sent to client in a json message inside label “data” with the same status code

If the received response code is 404, that means the product requested by the client is not present in catalog. The same status code is forwarded to client along with error message "product not found" in a json message, with label “error”.

For any other received response code, The status code “400” is sent to client along with error message "bad request” in a json message, with label “error”.

do_POST method:

The do_POST method handles all the post requests that frontend service receives.
It first checks if the parsed path starts with "/orders/", indicating it is a buy request which will go to order service. The client has sent an attached json message with name of product to be bought and requested quantity. The same json message is sent to the order service using "http://{ORDER_HOST}:{ORDER_PORT}/orders"
The response from the order service is stored in order_info variable. If the received status code is 200, that means order was successful and the received order details are sent to client in a json message inside label “data” with the same status code

If the received response code is 404, that means the product requested by the client is not present in catalog or is out of stock. The same status code is forwarded to client along with error message " product not found or is out of stock" in a json message, with label “error”.

For any other received response code, The status code “400” is sent to client along with error message "bad request” in a json message, with label “error”.

Catalog service

The catalog service uses inbuilt BaseHTTPRequestHandler, ThreadingHTTPServer from http.server library in python to handle incoming requests concurrently using threads. Thread is allotted to each incoming request from the inbuilt threadpool of ThreadingHTTPServer giving thread-per-request behavior. The mechanism chosen to design the interface here is HTTP REST. Synchronization is achieved using a global threading lock(threading.Lock()).

It starts on the specified port: 12501and host ip and listens for requests indefinitely using start_catalog_service method. (ip of the set up/machine where the service will run- can be modified according to set up used for running the application). Each time this microservice is started, it calls the load_catalog method inside start_catalog_service method.

load_catalog method:
This method acquires the global threading lock, and loads the previous state of the catalog that is stored in the disk as catalog.csv into the global catalog dictionary, that is initially empty. If the file does not exist, it creates this file and writes default values to the file and the global catalog dictionary.

handle_query method:
This method acquires the global threading lock, and accesses the catalog dictionary. It takes product name as input and searches for this product in the dictionary. If the product name is present in the catalog, its corresponding details are sent along with code 200, indicating the information fetching was successful, else just returns code 404.

handle_buy method:
This method takes received data from frontend server(which includes requested product name and quantity to be bought) as input.
If either of these parameters is none, it returns code 400, indicating bad request.

It then acquires the global lock and performs the following operations. It checks if the requested product name is present in catalog dictionary and has requested quantity<= corresponding quantity present in dictionary. If this condition is false, code 404 is retuned indication product is out of stock or is not found in catalog.

If this condition is true, it deducts the requested quantity from the catalog dictionary for the requested product name and writes the changes in the disk order.csv file. (updates in catalog are immediately written in disk file), and returns code 200, indication order can be processed.

CatalogRequestHandler class handles all the request that come to the catalog server.

The do_GET method:
This method handles all the get requests that catalog service receives.
At the end of the parsed path, the name of product resides for which corresponding details like quantity and price are requested. This product name is extracted and sent as input to handle_query method. The response code received as an output from handle query is sent to the front end server. If the response code from handle_query is 200, it attaches the product info received from handle_query in json format and sends both to frontend service.

The do_POST method:
This method handles all the post requests that catalog service receives.
The order service has sent an attached json message with name of product to be bought and requested quantity, this is stored in post_data variable. This post_data is sent as input to handle_buy method. The response code received as an output from handle_buy is sent to the order service.

Order service

The order service uses inbuilt BaseHTTPRequestHandler, ThreadingHTTPServer from http.server library in python to handle incoming requests concurrently using threads.
The mechanism chosen to design the interface here is HTTP REST. Thread is allotted to each incoming request from the inbuilt threadpool of ThreadingHTTPServer giving thread-per-request behavior. Synchronization is achieved using a global threading lock(threading.Lock()).

It starts on the specified port: 12502 and host ip and listens for requests indefinitely using start_order_service method. (ip of the set up/machine where the service will run- can be modified according to set up used for running the application). Each time this microservice is started, it calls the load_order_number method inside start_order_service method.

load_order_number method:
This method acquires the global threading lock, and loads the latest order number of order_log.csv that is stored in the disk into the global order_number variable and increments it. If the file does not exist, it initializes the global order_number to 0.

generate_order_number method:
This method acquires the global lock, returns the current order_number and increments it by 1. It is used to generate and return a unique consecutive order_number.

log_order method:
This takes order_number, product name, quantity as input. With lock it writes the order details in order_log.csv file in disk(if file does not exist, it is created and the received details are written in it.)

OrderRequestHandler class handles all the request that come to the order server.

The do_POST method:
This method handles all the post requests that order service receives.
The front_end service has sent an attached json message with name of product to be bought and requested quantity, this is stored in post_data variable. This post_data is sent to catalog service with post request. The response received from catalog service is saved in catalog_response variable.

If the status code received from catalog is 200, indicating order can be processed further as requested item is in stock, and catalog has been updated according to the requested quantity. The

order number is generated by calling generate_order_number method. The generated order number, product name and quantity is written to order_log disk by calling log_order method. The order number is sent to the frontend service in a json message, along with response code 200.

Else whatever response code was received from catalog service is sent to front end service.

Client.py

A session is created at client side to communicate with the front end service, to implement thread per session model. Sessions use keep-alive property and reuse the same connection to send all subsequent requests sent using this session.

In a for loop with 50 iterations,
Client sends a query request for a random product name from products list which has toy names to frontend service using
session.get(f"http://{FRONTEND_HOST}:{FORNT_END_PORT}/products/{product}"). The received json response is printed. If the get query was successful (status code =200) received quantity for the product is more than 0, with p probability, a buy request is sent with the same product name and quantity selected randomly between (1,50) using
order_response = session.post(f"http://{FRONTEND_HOST}:{FORNT_END_PORT}/orders/", json=order_data)
The received response for the order request is printed.

PART 2

For containerizing our application, we used docker. As we are required to run the application on docker, so we take the frontend host, order host, catalog host, frontend port, order port, catalog port all dynamically. This ensures that on whichever set up our client and servers run, the application uses those setup and ports. We did so by setting the environment variables: ORDER_HOST, CATALOG_HOST, FRONTEND_HOST, FRONTEND_LISTENING_PORT, ORDER_LISTENING_PORT, CATALOG_LISTENING_PORT.
We create the dockerfiles for all the 3 services: src/frontend.Dockerfile, src/order.Dockerfile, src/catalog.Dockerfile. We then have a shell script build.sh which contains a set of commands to build docker images, containers and run them. We ensure that all our docker containers run on the same network. We also mount a directory on the host as a volume to the catalog and order services so that files and output can persist after the containers are removed.

We the call ./build.sh script which runs the set of commands to start the microservices container. Once the services are started we run our client from our local machine.
We call ./build.sh to stop and delete the containers that we have created.
Note that we have used edlab for running our microservices and our local machine for running clients.
Additionally, we have also added a docker-compose file (docker-compose.yml) that can bring up or tear down all three services. We can just simply run the .yml file to build/tear down our containers. The file takes the environment variables from the .env file to run the container.
We run the microservices and our client and it works as expected. We can run ‘n’ number of clients. The .csv data files (catalog.csv and order_log.csv) get updated as we run our application.

LATENCY CALCULATION:
For latency calculation we measure the start time just as the session starts and measure the end time once our queries are all handled, before closing the session. We do our calculations for two scenarios: (a) When we run microservices and clients on different setups without docker and (b) When we run microservices and clients on different setups using docker. We vary our clients from 1 to 5 for both the scenarios and take the average of the response time and plot avg latency vs number of clients graphs for calculations. See the latency evaluation document for our calculations.

UNIT TESTING:
We write unit test cases testing/uniTests.py to see if the functionality of the test cases work as expected. We write test cases to see if the standalone microservices work as expected and also application as whole work as expected. We have tested the following use cases: query existing product from frontend service, query existing product from catalog, buy product from catalog, buy non existing product from catalog, place order from order, check if quantity of toy available from order, place order for non existing product from order, query non existing product from front end. The results can be found in the uniTestsOutput file.