677 Lab2 – Design Document

Part1 – Front-end Service

Design choices:
This server will implement a simple HTTP server using the thread per session model that can handle requests from multiple clients simultaneously using a thread pool. The server will be designed using Python and the sockets library, and will support HTTP 1.1.

Server Components:

- clientHandler() –
  a) This class implements methods required to communicate with the client and the backend services. The class has functions which parse the input http requests to extract the method and endpoint and route the request appropriately to the backend servers.
  b) The class also has a function which generates a http compliant message from the responses received from the backend servers.
  c) The communication with the backend servers is done using the **http requests** library.
- Httpserver - The server works on a thread per session model. It creates a thread pool of a given size(10). It implements method namely ***run*** to listen to incoming connections on a specified port using a socket. When a new client connection is received, a new instance of **clientHandler** is created with the **client_address** and **client_socket** arguments. This instance is then submitted to the thread pool. Each thread will be responsible for receiving, processing and sending back all responses from a given client.

API Endpoints

- Endpoint*: **/stocks/<stockname>***
  Description: Looks up for a stock
  Method: GET
  Request body: {}
  Response body:
  Success : {"data": {"name": string , "price" : float, "quantity": integer}}
  Error message : {"error": {"code": 404,"message": "stock not found"}}

- Endpoint: ***/orders***
  Description: Trade a stock
  Method: POST
  Request body: {"name": string, quantity": integer, "type": string}
  Response body:
  Success : {"data": {"transaction_number": integer}}
  Error messages :
  {"error": {"code": 404,"message": "stock not found"}}
  {"error": {"code": 403,"message": "Not enough stock"}}

Interfaces:

1) GET: ***/stocks/<stockname>***
The processRequest() function parses to extract the endpoint and stock name and sends the stockname to CatalogService() function where the service sends a http get request to the catalog server.
2) POST ***/orders/***
The processRequest() function parses to extract the endpoint and json body and sends the json body to the CatalogService() function where the service sends a http post request to the order server.

Part1 – Backend Service Catalog Server

Design choices:

- The catalog server interface is implemented using the HTTP REST API.
- The server is created using the `http.server` module in Python.
- The server is internally threaded and can handle multiple concurrent requests.
- A custom handler class `CatalogRequestHandler is used to handle incoming requests which inherits from the base `http.server.BaseHTTPRequestHandler` class
- The Http server is started with an object of CatalogService and CatalogRequestHandler as parameters.

Components:

- CatalogRequestHandler() : This class overwrites the `do_GET` and `do_POST` methods and calls a custom class to handle specific actions requested by the client.

- CatalogService() : The constructor of this class calls the **load_data()** function which initially loads data from stockCatalog.csv and stores it in a dictionary called **catalog.** This class is responsible for implementing all functions required to perform trade operations namely **lookup** and **trade**. This class also has a **backup_data** function which is started as a separate thread when the object of this class is created. The function periodically(every 3 seconds) backs up changes from the dictionary back to the csv. The object of this class is created and passed as a parameter for the server to start.

Synchronization:

- cataloglock : This lock is used by trade,lookup and backup_data functions while updating and looking up for stocks in **`catalog** dictionary.
- orderlock : This lock is used by backup_data() to take a lock on the stockCatalog.csv and periodically back data into it.

Datastructures/files:

- catalog: dictionary which stores the stockname, quantity, volume traded and price.
- stockCatalog.csv:Stores the stocks,quantity,volume traded and price.

API Endpoints

- Endpoint: ***/lookup/<stockname>***
  Description: Looks up for a stock
  Method: GET
  Request body: {}
  Response body:

Success : {"data": {"name": stockname, "price": price, "quantity": quantity}}
Error: {"error": {"code": 404, "message": "stock not found"}}

- Endpoint: ***/update/***
  Description: Updates the quantity and traded volume for the given stock
  Method: POST
  Request body: {'name':stockname,'quantity':reduced_quantity,'traded':quantity}}
  Response body:
  Success {"data": {"message":"Success"}}
  Error: {"error": {"code": 408, "message": 'Wrong endpoint'}}

Interfaces:

1) GET **/lookup/<stockname>**
   - The GET request is parsed in do_GET method inside the `CatalogRequestHandler to extract the API endpoint. The incoming requests for this endpoint are parsed to extract the stock name. The stock name is passed into the **lookup**() function of the CataogService class. The function takes a **cataloglock** on the **catalog** dictionary,looks up for the stock and sends a dictionary with values if the stock exists or sends an error message. The body is converted into json format and sent back as a http response.
2) POST **/update**
   - The incoming requests at this endpoint are parsed to get the dictionary body. The dictionary is passed to the **trade** function of CatalogService class. The function takes a lock on the **catalog** dictionary, updates the quantity and traded volume of the stock and returns a 'success' message as a dictionary. The body is converted into json format and sent back as a http response.

Part 1:Backend Service Order Server

Design choices:

- The order server interface is implemented using the HTTP REST API.
- The server is created using the `http.server` module in Python.
- The server is internally threaded and can handle multiple concurrent requests.
- A custom handler class `OrderRequestHandler which inherits from the base `http.server.BaseHTTPRequestHandler` class is used to handle incoming requests.
- The Http server is started with an object of orderService and OrderRequestHandler as parameters.
-

Components:

- OrderRequestHandler() : This class overwrites the **`do_POST** method and calls a custom class to handle specific actions requested by the client.

- orderService() : The constructor of this class initially assigns the **`count** variable to 0 before the server is started. This variable is incremented after every successful transaction. The class contains the **trade** function which is responsible for communicating with the catalog server and successfully complete a trade operation. The object of orderService is created before starting the server.

Synchronization:

- writelock: This lock is taken before ordercatalog.csv file is written to whenever there is a successful trade transaction. The write lock is taken and the transaction number,stock name,quantity and type of the stock are appended to the csv file.

- transactionlock: The **count** variable is locked using the transaction lock before incrementing it after a trade request is successful. The lock is taken, the count incremented,the writelock is taken, the ordercatalog is written to and then the transaction lock is released.

Datastructures/variables/files:

- count : variable which is used to keep track of the transaction number.
- ordercatalog.csv : Stores the transaction number,stock name,quantity and type

API Endpoints

- Endpoint: ***/trade/***
  Description: Sends a lookup request and then an update request to the catalog server
  Method: POST
  Request body: {"name": stockname, quantity": quantity, "type": buy/sell}
  Response body:
  Success {"data": {"transaction_number": count}}
  Error:
  {"error": {"code": 404, "message": "stock not found"}}
  {"error": {"code": 403, "message": 'Not enough stock'}}
  {"error": {"code": 405, "message": 'Unknown error retry'}}

Interfaces:

1) POST **/trade**

- The incoming requests at this endpoint are parsed to get the dictionary body. The dictionary is passed to the **trade** function of orderService class. The function sends a http **GET /lookup/<stockname>** request to the catalogServer.
- If the trade is a buy request, it checks the quantity returned and sends a **POST /update** request to the catalogServer with a json body that contains the stockname,actual quantity to update and the traded volume only if the returned quanity is greater than 0.If the trade is a sell request it directly sends it.
- If the transaction is successful, it takes a **`transactionlock**, increments the **`count** variable, takes the **`writelock**, updates ordercatalog.csv and returns the transaction number as a dictionary.

Client

Design choices:
This client will be designed using Python and the sockets library, and will support HTTP 1.1. The value of p is assigned to 0.5 for sending order requests.
Client components:

1) connect()-

- The function runs a loop 1000 times. Each time it picks a random stock from the list of stocks present and sends a http compliant GET request using sockets to the frontend service.
- The http response is decoded in **decode_http_response** to extract the json body and find if the lookup was successful or had an error.
- If the lookup was successful,a trade request body is generated using **create_trade_request** for the given stock and a http compliant post request is sent to the frontend service with a probability p.

2) create_trade_request()-

- The function generates trade dictionary by picking random value for quantity and choosing one of ‘buy’ or ‘sell’.

3) decode_http_response()

- The http response is split and parsed to find the start of the json body. The json body is loaded and a dictionary is returned.