Christopher Shi
33061961
cshi@umass.edu

Lab 2 Design Document
CS677 Distributed and Operating Systems

2023-03-25

1 Micro-level Implementations:

1.1 Interface/API Inputs and Outputs:

- Catalog Service:
  - Lookup:
    - input: stub containing the name of the stock to be looked up.
    - output: return (stock price, stock quantity) of the stock with the given name.
      return (stock price = -1, stock quantity = -1) if the stock with given name is not found.
  - Trade:
    - input: stub containing (stock name, stock trade quantity, trade type).
    - output: return 1 for successful trade.
- Order Service:
  - Request:
    - input: stub containing (stock name, stock trade quantity, trade type).
    - output: return transaction number.
      return -1 if stock with given name is not found.
      return -2 if the trade type is not ”buy” or ”sell”.
      return -3 if the quantity of stocks to trade is not valid (negative number. non-ints are not considered since PROTO already defines that quantity has to be an int.)
      return -4 if the trade type is ”buy” and the quantity to be bought is greater than the quantity of stock available.
- Front end service:
  - GET:
    - input: ”GET /stocks/stockName”
    - output: return data json with name, price, and quantity {data:{name, price, quantity}}.
      return error json with {error:{’code’:404,’message’:”stock not found”}} when stock with given name not found.
  - POST:
    - input: ”POST /orders bodyJSON” (where bodyJSON contains json with name, quantity, type {name, quantity, type})
    - output: return data json with transaction number {data:{transaction_num}}.
      return error json with {error:{’code’:404,’message’:”stock not found”}} when stock with given name not found.
      return error json with {error:{’code’:400,’message’:”invalid request type”}} when the trade type is not ”buy” or ”sell”.
      return error json with {error:{’code’:404,’message’:”invalid number of stocks”}} when the quantity of stocks to trade is negative.
      return error json with error = {error:{’code’:404,’message’:”not enough stocks available to buy”}} when the amount of stocks to buy is greater than the quantity available.

1.2 Order Service Implementation:

- Database file (JSON format): database file is written as a dict of transaction numbers which are each mapped to a dict containing (name, type, quantity). This allows for easy writing and reading. At program start, the program checks if a database file exists in current folder, if it does, load the json to the in memory dict/database.

```
        #Check if database file exists in current folder, if it does load the json to a dict
        if os.path.exists('./order.txt'):
            with open('./order.txt') as file:
                transactions = json.loads(file.read())
            #Continue the transaction number by finding the maximum key in the dict
            transactionNum = int(max(transactions, key=int))
        else:
            transactions = {}
            transactionNum = -1
```

- The order server is always hosted on port 56891 s.t. it is easy to find. The server utilizes thread pool executor with a pool size of 5. This yields a thread per request architecture. Locking is not utilized since threadpoolexecutor utilizes GIL. All error handling associated with a trade request is handled on the order service. On a trade request, the order server does the following:

  1. Create a stub and connect to the catalog service (port 56892).
  2. Call the lookup function of the catalog service.
  3. If the returned stock price and stock quantity are both -1, return -1. (catalog service returns (-1,-1) when the stock cannot be found.)
  4. If the trade type is not ”buy” or ”sell”, return -2.
  5. If the amount of stock to trade is less than 0, return -3.
  6. If the trade type is ”buy”, and the quantity is greater than the quantity available (quantity known after calling lookup from catalog in step 2)
  7. If no errors are called first, create another stub and call the trade function of the catalog service (port 56892). (s.t. we can update the quantity and volume of the stock) Iterate the transaction number. Map the transaction number to a dict containing the name of the stock traded, the trade type, and the quantity of stock traded. Write the current in memory dict/database as a json to a txt file. return the current transaction number.

```
            #Takes the string stockName and returns the price of the stock and the
                trading volume so far.
            def Request(self, request, context):
                global transactions
                global transactionNum
                tradeType = request.tradeType
                stockName = request.stockName
                stockTradeQuantity = request.stockTradeQuantity
                #Connect to the catalog server
                with grpc.insecure_channel(cataloghost + ':56892') as channel:
                    stub = stockbazaar_pb2_grpc.CatalogStub(channel)
                    #Lookup the given stock
                    res = stub.Lookup(stockbazaar_pb2.lookupStockName(stockName =
                        stockName))
                    #Stock not found
                    if res.stockPrice == -1 and res.stockQuantity == -1:
                        return stockbazaar_pb2.requestResult(transactionNum = -1)
                    #Invalid request i.e. not "buy" or "sell"
                    elif tradeType not in ['buy','sell']:
                        return stockbazaar_pb2.requestResult(transactionNum = -2)
                    #Trying to buy or sell an invalid number of stocks
                    elif stockTradeQuantity < 1:
                        return stockbazaar_pb2.requestResult(transactionNum = -3)
                    #Trying to buy more stocks than available
                    elif tradeType =='buy' and stockTradeQuantity > res.stockQuantity:
                        return stockbazaar_pb2.requestResult(transactionNum = -4)
                    else:
                        #Connect to the catalog server s.t. it can update quantity and
                            volume
                        res = stub.Trade(stockbazaar_pb2.tradeStockName(stockName =
                            stockName, stockTradeQuantity = stockTradeQuantity,
                            tradeType = tradeType))
                        transactionNum += 1
                        #Write to database in memory
                        transactions[transactionNum]={'name':stockName,'type':tradeType,
                            'quantity':stockTradeQuantity}
                        #Write the current order database to a txt file in json format
                        with open('./order.txt', 'w') as file:
                            file.write(json.dumps(transactions)) # use `json.loads` to
                                do the reverse
                        return stockbazaar_pb2.requestResult(transactionNum =
                            transactionNum)
```

1.3 Catalog Service Implementation:

- Database file (JSON format): database file is written as a dict of company names mapped to a dict containing (price, volume, quantity). At program start, the program checks if a database file exists in current folder, if it does, load the json to the in memory dict/database. Else, initialize the in memory dict/database with some default values.

```
        #Load the database json txt if it exists
        if os.path.exists('./catalog.txt'):
            with open('./catalog.txt') as file:
                stockData = json.loads(file.read())
        else:
            #Create initial on disk data file
            stockData = {}
            tesla = {'price':183.26,'volume':0,'quantity':1000}
            ford = {'price':11.93,'volume':0,'quantity':1000}
            apple = {'price':152.59,'volume':0,'quantity':1000}
            amazon = {'price':94.88,'volume':0,'quantity':1000}
            nvidia = {'price':240.63,'volume':0,'quantity':1000}
            intel = {'price':28.01,'volume':0,'quantity':1000}
            meta = {'price':194.02,'volume':0,'quantity':1000}

            stockData['apple']=apple
            stockData['tesla']=tesla
            stockData['ford']=ford
            stockData['amazon']=amazon
            stockData['nvidia']=nvidia
            stockData['intel']=intel
            stockData['meta']=meta
            with open('./catalog.txt', 'w') as file:
                file.write(json.dumps(stockData)) # use `json.loads` to do the reverse
```

- The catalog server is always hosted on port 56892 s.t. it is easy to find. The server utilizes thread pool executor with a pool size of 5. This yields a thread per request architecture. Locking is not utilized since threadpoolexecutor utilizes GIL. On lookup request, the catalog server does the following:

  1. Get the dict associated with the current stock name from the in memory dict/database.
  2. If the length of the dict returned is 0, we know that the inputted stock name does not exist. return (stockprice =-1, stockquantity=-1) to denote that the stock was not found.
  3. Else return (stockprice, stockquantity) as read from the dict returned from step 1.

```
            #Takes the string stockName and returns the price of the stock and the
                trading volume so far.
            def Lookup(self, request, context):
                global stockData
                #Get stock price for the given stock name
                stockInfo = stockData.get(request.stockName, {})
                #If the name cannot be found in dict, return -1/-1
                if len(stockInfo) == 0:
                    return stockbazaar_pb2.lookupResult(stockPrice=-1,stockQuantity=-1)
                #If found, return stock price and stock volume
                else:
                    return stockbazaar_pb2.lookupResult(stockPrice=stockInfo['price'],
                        stockQuantity=stockInfo['quantity'])
```

  On trade request, the catalog server does the following:

  1. If the trade type is ”buy”, decrement the stock quantity by the trade quantity. Increase the stock volume by the trade quantity.
  2. If the trade type is ”sell”, increase the stock quantity by the trade quantity. Increase the stock volume by the trade quantity.
  3. Write the current in memory dict/database as a json to a txt file. return 1 signifying success.

```
             #Buys or sells N items of the stock and increments the trading volume of
                 that item by N.
        def Trade(self, request, context):
            global stockData
            tradeType = request.tradeType
            stockName = request.stockName
            stockTradeQuantity = request.stockTradeQuantity

            #If the trade type is Buy, increment stock volume
            if tradeType == "buy":
                (stockData[stockName])['quantity'] -= stockTradeQuantity
                (stockData[stockName])['volume'] += stockTradeQuantity
            #If the trade type is Sell, increment stock volume
            elif tradeType == "sell":
                (stockData[stockName])['quantity'] += stockTradeQuantity
                (stockData[stockName])['volume'] += stockTradeQuantity
            with open('./catalog.txt', 'w') as file:
                 file.write(json.dumps(stockData)) # use `json.loads` to do the reverse
            #Successfully buy/sell returns 1
            return stockbazaar_pb2.tradeResult(resultTrade = 1)
```

1.4 Front End Implementation:

- Front end HTTP connections are implemented with http.server and BaseHTTPRequestHandler. Thread pool is created through ThreadingMixIn package and ThreadPoolExecutor. Thread per session approach is implemented by ensuring that headers sent between the front end service and clients contain ”Connection: keep-alive”. The front end service is always hosted on port 56893 s.t. it is easy to find. On GET request, the front end service does the following:

  1. Create a stub and connect to the catalog service. (port 56892)
  2. Call the lookup function of the catalog service.
  3. If stockprice=-1 and stockquantity=-1 create an error dict {error:{’code’:404,’message’:”stock not found”}} when stock with given name not found. Send a response code of 404.
  4. Create a data dict containing {data:{name,price,quantity}}. Send a response code of 200.
  5. Send a header with ”Connection: keep-alive” and ”Content-length: size of dict”
  6. Convert the error/data dict to a json, encode the json, and write response.

```
         def do_GET(self):
        with grpc.insecure_channel(cataloghost + ':56892') as channel:
            stub = stockbazaar_pb2_grpc.CatalogStub(channel)
            #Extract the name of the stock from the HTTP request
            stockName = self.path.split('/')[2]
            res = stub.Lookup(stockbazaar_pb2.lookupStockName(stockName = stockName))
            if res.stockPrice == -1 and res.stockQuantity == -1:
                #send code 404 response
                self.send_response(404)

                response = {}
                error = {'code':404,'message':"stock not found"}
                response['error']=error

            else:
                #send code 200 response
                self.send_response(200)

                response = {}
                data = {'name':stockName,'price':round(res.stockPrice,2) ,'quantity':res
                    .stockQuantity}
                response['data']=data

            json_data = json.dumps(response)
            #send header first
            self.send_header("Connection", "keep-alive")
            self.send_header("Content-Length", str(len(bytes(json_data, 'utf-8'))))
            self.end_headers()

            self.wfile.write(json_data.encode())
        return
```

  On POST request, the front end service does the following:

  1. Create a stub and connect to the order service. (port 56891)
  2. Read the ”content-length” header of the POST request, and read that many of bytes of data from the input s.t. we are able to parse the json contaning the body.
  3. The bytes are converted to a dict object, and the request function from the order service is called, reading the quantity to be traded and the trade type from the input dict.
  4. If the returned transaction number = -1, create an error dict {error:{’code’:404,’message’:”stock not found”}} when stock with given name not found. Send a response code of 404.
  5. If the returned transaction number = -2, create an error dict {error:{’code’:400,’message’:”invalid request type”}} Send a response code of 400.
  6. If the returned transaction number = -3, create an error dict {error:{’code’:404,’message’:”invalid number of stocks”}} Send a response code of 400.
  7. If the returned transaction number = -4, create an error dict {error:{’code’:404,’message’:”not enough stocks available to buy”}} Send a response code of 400.
  8. If no error is returned, create a data dict containing {data:{transaction_num}}. Send a response code of 200.
  9. Send a header with ”Connection: keep-alive” and ”Content-length: size of dict”
  10. Convert the error/data dict to a json, encode the json, and write response.

```
        def do_POST(self):
        with grpc.insecure_channel(orderhost + ':56891') as channel:
            stub = stockbazaar_pb2_grpc.OrderStub(channel)
            #Read input json
            content_length = int(self.headers['Content-Length'])
            inputData = self.rfile.read(content_length)
            #Convert bytes to a dict object
            inputData = json.loads(inputData)
            stockName = inputData['name']

            res = stub.Request(stockbazaar_pb2.requestStockName(stockName = stockName,
                stockTradeQuantity = inputData['quantity'], tradeType = inputData['type'
                ]))
            res = res.transactionNum
            #Stock not found
            if res == -1:
                #send code 404 response
                self.send_response(404)

                response = {}
                error = {'code':404,'message':"stock not found"}
                response['error']=error
            #Invalid request i.e. not "buy" or "sell"
            elif res == -2:
                #send code 400 response
                self.send_response(400)

                response = {}
                error = {'code':400,'message':"invalid request type"}
                response['error']=error
            #Trying to buy or sell an invalid number of stocks
            elif res == -3:
                #send code 400 response
                self.send_response(400)

                response = {}
                error = {'code':404,'message':"invalid number of stocks"}
                response['error']=error
            #Trying to buy more stocks than available
            elif res == -4:
                #send code 400 response
                self.send_response(400)

                response = {}
                error = {'code':404,'message':"not enough stocks available to buy"}
                response['error']=error
            else:
                #send code 200 response
                self.send_response(200)

                response = {}
                data = {'transaction_number':res}
                response['data']=data

            json_data = json.dumps(response)

            #send header first
            self.send_header("Connection", "keep-alive")
            self.send_header("Content-Length", str(len(bytes(json_data, 'utf-8'))))
            self.end_headers()

            self.wfile.write(json_data.encode())
            return
```

1.5 Client Implementation:

- Client HTTP connections are implemented with http.client package. The ip address, port number, probability, and verbosity are configured as command line arguements. The client does the following 100 times:

  1. Select a stock name from: [”tesla”,”ford”,”apple”,”amazon”,”nvidia”,”intel”,”meta”,”imaginarycompany”] which represents 7 real stocks, and one stock to test for when an unknown stock is fed to the server.
  2. Send a GET request to the front end service (port 56893)
  3. Sample a number from [0,1]. If the number if greater than probability:
     (a) Randomly choose from [”buy”, ”sell”,”trade”].
     (b) Randomly choose an int from [-10,100]
     (c) Write the stock name, trade type, quantity, to a json object and send a POST request to the front end service with the body equal to the json object as well as a header containing the size of the json body.

```
        #get http server ip
http_server = sys.argv[1]
```

```
port = sys.argv[2]
probability = float(sys.argv[3]) #Set the probability p at which it will send another order request using the same connection.
mode = int(sys.argv[4]) #0 for verbose 1 for non-verbose/only latency shows
#create a connection
conn = http.client.HTTPConnection(http_server, port)
stockNames = ["tesla","ford","apple","amazon","nvidia","intel","meta","imaginarycompany"]
start_time = time.time()
#Running total of number of requests sent s.t. we can calculate average latency
numRequests = 0
for _ in range(100):
    stockName = random.choice(stockNames)
    if mode == 0: print("-------------- Lookup --------------")
    if mode == 0: print("Input: " + stockName)
    conn.request('GET','/stocks/' + stockName)
    #get response from server
    rsp = conn.getresponse()
    data_received = rsp.read()
    #Convert data to dict
    data_received = json.loads(data_received)
    numRequests += 1
    if mode == 0: print("Output: " + str(data_received))

    order = random.uniform(0,1)
    if order > probability:
        if mode == 0: print("-------------- Trade --------------")
        data = {}
        data['name']=stockName
        #Randomly choose to buy or sell or invalid option
        tradeTypes = ["buy", "sell","trade"]
        tradeType = random.choice(tradeTypes)
        data['type']=tradeType
        #Randomly select how many stocks to buy sell within range of 1 to 100
        stockQuantity = random.randint(-10,100)
        data['quantity']=stockQuantity
        json_data = json.dumps(data)
        if mode == 0: print("Input: " + str(json_data))
        conn.request('POST', '/orders', body = json_data.encode(), headers={'Content-Length':str(len(bytes(json_data, 'utf-8')))})
        #get response from server
        rsp = conn.getresponse()
        data_received = rsp.read()
        #Convert data to dict
        data_received = json.loads(data_received)
        numRequests += 1
        if mode == 0: print("Output: " + str(data_received))
    if mode == 0: print("\n")
print("Average Latency: " + str((time.time() - start_time)/numRequests))
conn.close()
```

1.6 Docker Specific Implementation Details:

- Given that when connecting to servers with stubs, the format is a string containing "ip_address:port_number", we need some way to tell each service what the IP address of the other services are. As such, we configure the IP addresses of each other service as an environmental variable. If no environmental variable is given, the services default to the ip address that they are being hosted on.

```
    #Order Service
    if __name__ == '__main__':
        cataloghost = os.getenv("CATALOG_HOST", socket.gethostbyname(socket.gethostname()))
```

```
    #Front End Service
    if __name__=="__main__":
        cataloghost = os.getenv("CATALOG_HOST", socket.gethostbyname(socket.gethostname()))
        orderhost = os.getenv("ORDER_HOST", socket.gethostbyname(socket.gethostname()))
```

- We hard code the IP assignments for each service in the docker compose file. We hard code the port numbers assigned to each service in the docker compose file. We pass the IP addresses of the other services as environmental variables in the docker compose file. We establish a network subnet of 10.0.0.0/24 s.t. all IP addresses of the form 10.0.0.xxx can connect to our service as long as the port number is 56893. For the catalog and order services, their respective subfolders are mounted s.t. the local database txt files are able to persist even after docker closure.

```
    #version: "3.8"
services:
  order:
    build:
        context: .
        dockerfile: ./DockerFile-order
    ports:
      - "56891:56891"
    volumes:
      - ./order-service:/app/order-service
```

```
    environment:
      - CATALOG_HOST=10.0.0.242
    networks:
      vpcbr:
        ipv4_address: 10.0.0.241
  catalog:
    build:
        context: .
        dockerfile: ./DockerFile-catalog
    ports:
      - "56892:56892"
    volumes:
      - ./catalog-service:/app/catalog-service
    networks:
      vpcbr:
        ipv4_address: 10.0.0.242
  frontend:
    build:
        context: .
        dockerfile: ./DockerFile-frontend
    ports:
      - "56893:56893"
    environment:
      - CATALOG_HOST=10.0.0.242
      - ORDER_HOST=10.0.0.241
    networks:
      vpcbr:
        ipv4_address: 10.0.0.243
networks:
   vpcbr:
     driver: bridge
     ipam:
      config:
        - subnet: 10.0.0.0/24
```

- There are three separate dockerfiles created, each corresponding to one of the services in the system. They all contain the same form, mapping their respective subfolders to the working directories, as well as copying their respective subfolders to their working directories.

```
FROM python:3.8-alpine

RUN pip install flask redis grpcio grpcio-tools

WORKDIR /app/catalog-service

COPY ./catalog-service /app/catalog-service

ENTRYPOINT ["python", "-u", "catalogServer.py"]
```

2 Macro-level Implementations:

2.1 System Design

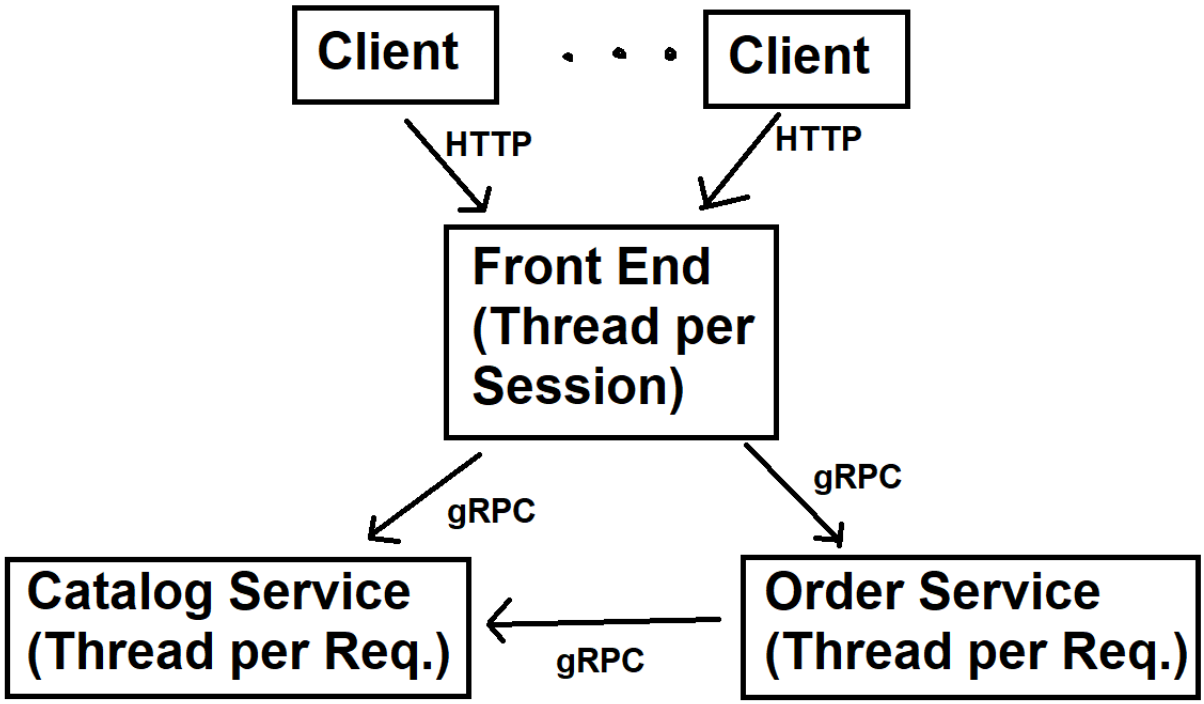

Figure 1: Diagram showing how requests are sent in the stock server architecture. Requests between the client and the front end service are HTTP requests. Requests between the front end service, the catalog service, and the order service, are gRPC requests.

- The catalog service will always be on port 56892. The order service will always be on port 56891. The front end service will always be on port 56983.
- All services are hosted on the local ip of the host machine when not utilizing docker. Clients connect to the front end service by connecting to the public IP of the host machine on port 56893.
- When utilizing docker, the catalog service is hosted on 10.0.0.242, the order service is hosted on 10.0.0.241, and the front end service is hosted on 10.0.0.241. Clients connect to the front end service by connecting to the public IP of the host machine on port 56893.

2.2 http.client/http.server:

- The http.client and http.server packages are utilized in order to facilitate HTTP connections between the clients and the front end microservice. GET and POST requests are sent to the front end service. GET and POST are specifically implemented on the front end service. One thing to note is that ”Connection: keep-alive” is sent from both the client, as well as a response from the front end service. This is in order to ensure that the same connection is utilized per session (i.e. thread per session).

2.3 gRPC:

- gRPC is utilized to handle the following requests:
  - Front end to catalog service.
  - Front end to order service.
  - Order service to catalog service.

In each service subfolder, a copy of the pb2 and pb2_grpc files are included in order to allow for gRPC to work properly.

2.3.1 proto File:

- Catalog service:
  - Lookup:
    - input: string containing the name of the stock
    - returns: float containing stock price. int containing stock quantity.
  - Trade
    - input: string containing the name of the stock. int containing number of stocks to trade. string containing the type of trade.
    - returns: int containing the result of the trade.
- Order service:
  - Request:
    - input: string containing the name of the stock. int containing number of stocks to trade. string containing the type of trade.
    - returns: int containing the result of the trade.