Design Document

Pyro NameServer:

LAB 3:
To run Pyro5 nameserver on a local machine, you can start it by running the **pyro5-ns** command in a terminal. For running Pyro5-ns on AWS, need to export pyro5 in path.

Catalog Service:

Lab3:

The catalog service after trade, additionally sends a invalidate cache post request to the front-end to remove the traded stock from the cache. It does by sending the invalidated stock's name to the front-end service which then removes it from the cache. This caching can be turned off/on using the environment variable CACHE in the .env file.

Lab2:

Catalog service is designed using RPC implemented using Pyro. We have made the server threaded by setting these commands. Using server type threaded assigns each incoming request to a thread from the thread pool. Pyro threaded server uses a dynamic thread pool and here we have specified the number of minimum threads to be maintained as 5.

```
#Making pyro threaded server
Pyro5.config.SERVERTYPE = "thread"
Pyro5.config.THREADPOOL_SIZE_MIN = 5
```

We have to store the quantity and the volume of the stock in a file as well in-memory. We first check if the file exists, if it does then we just load the file's data in memory. Otherwise we initialise the stock data ourself as it means that it is the first time running the catalog service. We initialize each stock with the volume 0 and quantity 100.

The CatalogService class does two things, it returns the lookup response and also updates the volume and quantity of the stock based on the trade request that has been sent by the orderservice. Therefore it has 2 methods, lookup which accepts a stock name, and trade which takes stock name, trade type and quantity from the order service.

We have implemented the trading logic in the catalog service which checks if the request quantity for buying a stock is less than or equal to the quantity mentioned in the stocks.json file. The reason we have done this in the catalog service is so that we can reduce the number

of calls to the catalog service. For eg: If this trading logic was implemented in the order service, then we would first call lookup and if lookup was a success then we would run the trading logic and again call the trade function. But in our case, we can reduce this to one call per client by forwarding the trade request to catalog.

Lab 2

Lookup

If the lookup function gets an invalid stock name, error code 402 is passed to the client stating stock not found else we return stock data with name price and available quantity.

Trade

In the trade function, first we get the trade type as 1 or -1 indicating selling and buying respectively as it helps in checking if the quantity sent is correct and updating the quantity as -1 will decrease the quantity and 1 will increase. Response is sent as 1, 0, -1 which indicates Success, invalid stock name, or not enough stocks respectively to the order service.

Both the functions have normal locks allocated to them, therefore while performing lookup which performs read or while performing trade which does write functions they should wait the lock to make sure there are no inconsistencies in data.

For registering the services on the Pyro5 server we have first started the daemon at the container's ip if it's running in a container or the localhost if it's running on localhost.

```
container_ip = socket.gethostbyname(socket.gethostname())
daemon = Pyro5.server.Daemon(host = container_ip)
```

After starting the daemon we have registered the services on nameserver which can be automatically located using Pyro as it is running on 0.0.0.0.

Order Service

Lab 3

There are multiple order services that needs to be running so the class will have a static variable by which it knows it's ID. This ID is passed through command line arguments while running the orderService.py file. As each orderService will have it's own log and pickle file, they are dynamically attached to the base file names and stored as static variables.

On start up we first check if the front end is running, if it is running it means there is a leader assigned and there might be transactions that happened when it was off. So we get the leader

from front end and get all the new transactions from the leader orderService. For this we sent the last transaction ID we have and get all the transactions after it. If front end doesn't exist, it means no leader is selected and no transactions have taken place therefore it starts as independent. The fetchTransactions function takes the ID as the leader and sends all transactions after it to the caller.

There is a getTransaction function which sends all the transactions for the transaction IDS that are sent to it. This is required for client to check if it's log file is the same as the orderService log.

The Trade function will only be called of the leader orderService, which on a successful transaction sends the log to all the follower orderServices to maintain consistency.

We get the front end PORT and the number of replicas from the environment variable file and can change the front end host on top of the file.

While registering the orderService on Pyro nameserver we attach the ID which makes it service.order3 if the ID is 3 on nameserver. These nameserver names are then stored inside the env file for front end to access them.

LAB2:

Order service is also designed by the principle of RPC using Pyro. It is also threaded similarly to the catalog service. Therefore it is a dynamic threadpool and we have set the threads as 5 on top of the file.

```
# Making pyro threaded server
Pyro5.config.SERVERTYPE = "thread"
Pyro5.config.THREADPOOL_SIZE_MIN = 5
```

The order service is responsible for getting orders from the front end which It forwards to the catalog service. It also stores the transaction number and the trade data for all the successful trades in tradeLog.csv file. Last transaction number should be stored outstide in-memory because if the code restarts we don't want the transaction number to start from 0. Therefore we have pickled the transaction number in a pickle file. If the pickle file does not exist that means it's the first time running the program which initialised the transaction number to 0.

After sending the order to the trade function to catalog service and getting a result, is the result is 1 it means the order is successful and therefore the log is added to the order log file which is updated after every successful order and the transaction number is increased by 1. A result of 0 send back error code 402 saying wrong stock name, and -1 sends error code 404 which means not enough stocks available to the front end service which sends it back to client.

A lock is implemented here which makes sure two order service cannot be made at the same time therefore two transactions cannot have the same transaction number.

For making it accessible across different container, the daemon, similar to catalog service is running on the container ip and is registered on the nameserver.

```
container_ip = socket.gethostbyname(socket.gethostname())
daemon = Pyro5.server.Daemon(host = container_ip)
# find the name server
ns = Pyro5.api.locate_ns()  #Locating nameserver
uri = daemon.register(CatalogService)
ns.register("service.catalog", uri) #Registring
```

Front End:

Lab 3

The front end does 2 new things, caching and managing replicas of OrderService.

For leader selection we need the number of replicas and uri of all orderServices so front end could talk to it. We get these using the common .env files. On startup we first check all the alive orderService replicas using the healthCheck function. If a orderService replica is down an Exception is caused which is handled and used to identify if replicas are alive or not.

The highest alive one is selected as the main leader and stored in global variables. A get service is made which just send the leader ID back to the services. All the incoming order requests are sent to the leader order service.

If the leader goes down, it is detected using try and except while making a new order which again triggers a leader election.

There is also a “logVal” get request which gets the log data for the transaction IDs from the orderService.

For caching, we have an in-memory dictionary which is updated everytime a lookup request is made. The stock information is stored in the form cache[“stock_name”] = stock_info. Each time a lookup request is made, the frontend service checks if the stock name is present in the cache, if so, it return the stock_info, else it fetches from the catalog service. The cache is invalidated through a post request sent by the catalog service each time a trade request is successful. On a post ‘/invalidate_cache’ request which contains the name of the stock, the front end service removes the stock information for that stock.

Lab 2

The front end is HTTPv1.1 based REST api which supports GET and POST functions for the client to connect. This is implemented using various libraries such as http.server and socketserver. To make the implementation use dynamic threadpool, socketservers ThreadedMixIn class is used which handles multiple clients simultaneously by using a threadpool executor.

The HTTP version used here is 1.1 which persists the connection of the client if they want to send another request over the same connection.

For the GET function, we have implemented a method do_GET() which gets the URL and parses it using urllib.parse. After getting the stock name, a remote object is created of the catalogservice using pyro nameserver which send the uri for creating the proxy. Then we contact the object as it's our own method which is definition of RPC. The json that is returned from the catalog service is directly sent back to the client through the wfile.write command().

For the POST function, we receive a json file from the client which has the order details such as stock name, quantity and type of order. The content is accessed using the below functions. And then again, a remote order service object is initialised using pyro5. The json that is returned from the order service is directly sent back to the client through the wfile.write command.

```
content_length = int(self.headers['Content-Length'])
orderData = json.loads(self.rfile.read(content_length).decode())
```

The host ip, port and number of threads are set at the top of the file.

```
HOST = "0.0.0.0"    #Running over all available ips
PORT = int(os.getenv("HOST_PORT"))#Loading from env variable
NUM_THREADS = 5 #Setting max threads
```

The port is set in the ".env" file which contains two environment variables "FRONTEND_PORT" and "HOST_PORT". There are some changes required to run depending on if you are running in a container or the local host. The frontend port variable is used when the front end is run inside a container as there will be port forwarding from the host port which is on the local machine to the front end port of the front end container. Therefore FRONTEND_PORT will be used here if the front end is running inside the container through docker-compose and HOST_PORT should be used here if it running on the host computer itself.

Client:

Lab 3

The client also makes it own log file which is a new file every time the client starts and appends on every successful order. The client has no idea of failure in orderService because it is completely handled by the front end.

After the client is done running, it gets all it's transaction ID from the log file and sends it to front end which returns the logs from the order. All the logs are then compared and are printed where True means that log is matching and false means its not.

Lab 2

The client is an automated client which sends requests to the front end server by selecting quantity, stock name and type of trade. This is done by using random functions which selects from these arrays.

```
TRADE_TYPES = ["buy", "sell"]
STOCK_NAME = ["GameStart", "BoarCo", "MenhirCo", "FishCo"]
```

The HOST, PORT and the probability for making a trade order can be changed on top of the file.

```
HOST = "localhost"
PORT = os.getenv("HOST_PORT")
PROBAB = 0.7      #Probability for making a trade order
```

The port can be changed through the .env file. This is the port of where the front end is running. Therefore it will always be HOST_PORT.

The client is implemented using http.client through which it first connects to the front end server. A GET request is made of the selected stock_name which gets back the price and quantity. After which with the probab set above, an POST order request is made of the same order which either returns transaction number or error message.

Testing:

Lab 3:

Using our test suite, we test the replication through test_Replication() by checking if the last log in all the trade logs is the same. We test the fault tolerance, by testing if the leader is elected at present in the test_LeaderElection(), and we test the cache by sending a loopup request and checking if it is present in cache and then sending a trade request and ensuring the stock is removed from the cache in test_cache.