Christopher Shi
33061961
cshi@umass.edu

Lab 3 Design Document
CS677 Distributed and Operating Systems

2023-04-28

1 Design Choices:

- Utilize HTTP GET /cache/ request as the server-push method for invalidation requests. Request sent from order service leader replica to the front end. Occurs after each successful trade s.t. the local cache of the front-end is always up to date.
- Each time a leader is elected at the front end, a novel AssignFollowers() function is called on the leader replica which populates a dict of followers.
- Leader replica ID is kept track of on the front end service.
- During each propagation call from the leader to its followers, the leader first checks if the follower can be pinged. (To account for follower crashes.)
- On successful trade made by the leader replica, the writing of data and the propagation of data are all locked. (Without a lock, sometimes different clients end up with the same transaction numbers.)
- When a crashed replica rejoins, it sends an HTTP GET /leader/ request to the front end service in order to get the ip address and port of the current leader replica.
- When a crashed replica rejoins, it is added to the list of followers of the current leader. No new leader election occurs even if the rejoining replica's ID is higher than the current leader's.
- During leader election, we always start from largest replica ID and work our way down. Each replica ID is pinged to ensure connection before being assigned as the leader.
- If caching is turned off, the order service leader replica will not send the HTTP GET /cache/ requests to the front end.
- POST and GET /orders/ requests are always locked. (Without a lock, multiple clients may start multiple leader elections processes. With the lock, only one leader election can occur at a time.)
- The front end in memory cache was implemented with a simple dictionary.
- If new transactions are added to the leader's database while a rejoined replica is being synchronized, the new transactions will trivially be synchronized with the rejoined replica. This will continue to occur as long as synchronization takes, at which point new transactions will simply be propagated to the rejoined replica. (We synchronize by simple updating the rejoined replica in a for loop with transaction info until we reach the maximum transaction number in the leader. If the leader is still servicing new requests, trivially the maximum transaction number will increase and the new requests will be synchronized with the rejoined replica.)
- Leader elections occur only at first boot, and if the current leader cannot be reached during POST requests and GET /orders/ requests.
- Clients store order information as a dict of {transaction number:{transaction number, stock name, trade type, trade quantity},transaction number:{transaction number, stock name, trade type, trade quantity}, ..., transaction number:{transaction number, stock name, trade type, trade quantity}}
- If the leader crashes during POST requests or GET /orders/ requests, leader election occurs, and the same request is carried out on the new leader. (No requests are lost even when a leader crashes.)
- For both propagation requests and successful trades on the leader, the physical databases are immediately updated after each request.

2 Micro-level Implementations:

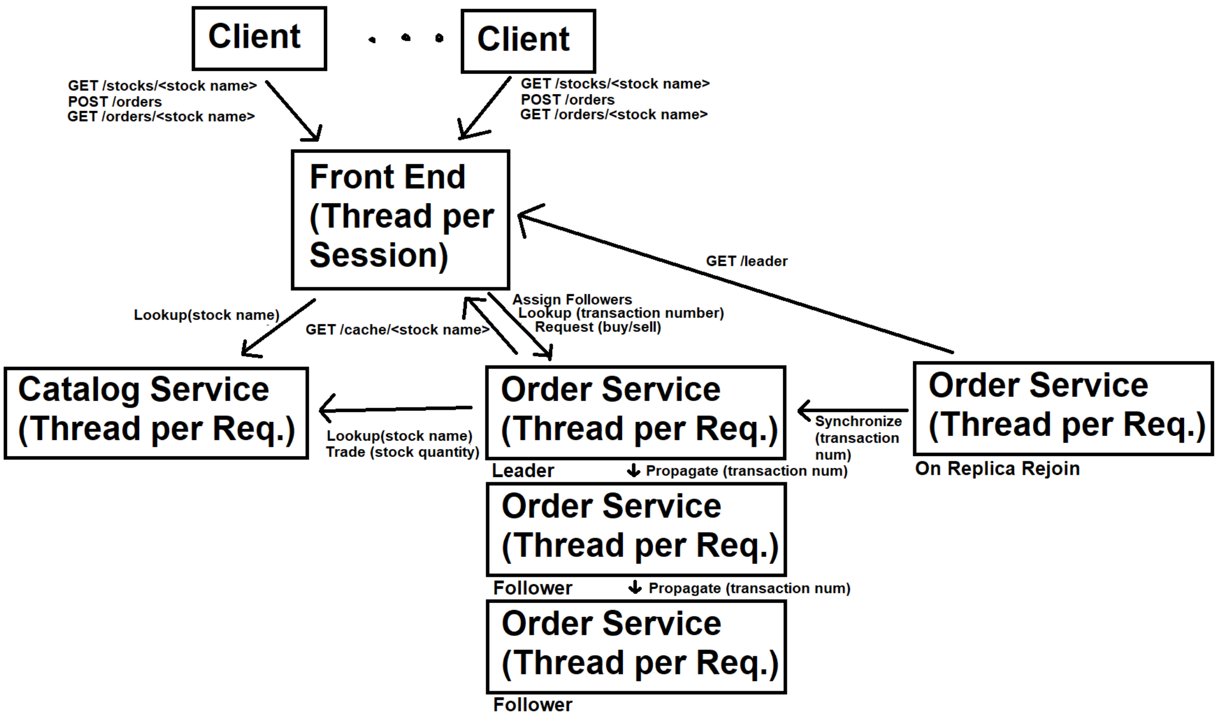

Figure 1: Diagram showing how requests are sent in the stock server architecture.

2.1 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.
  - Lookup:
    - ⋆ input: stub containing the transaction number of the transaction to be looked up.
    - ⋆ output: return (transaction number, stock name, trade type, trade quantity) of the transaction with the given transaction number.
      return (transaction number=-1, stock name=””, trade type=””, trade quantity=-1) if an invalid transaction number is given.

- Propagate:
  - input: stub containing (transaction number, stock name, trade type, trade quantity)
  - output: return 1 for successful propagation.
- AssignFollowers:
  - input: stub containing two addresses, in the form of (followerOne=address1, followerTwo=address2)
  - output: return 1 for successful follower assignment.
- Synchronize:
  - input: stub containing (latest transaction number of rejoining replica, address of rejoining replica)
  - output: returns a stream of transaction information, in the form of (transaction number, stock name, trade type, trade quantity).

- Front end service:
  - GET:
    - input: **"GET /stocks/<stock name>"**
    - 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.
    - input: **"GET /orders/<transaction number>"**
    - output: return data json with number, name, type, quantity {data:{number, name, type, quantity}}
      return error json with {error:{'code':404,'message':"transaction not found"}} when transaction with given transaction number not found.
    - input: **"GET /cache/<stock name>"**
    - output: return an empty data json {data:{}}
    - input: **"GET /leader"**
    - output: return data json with leader {data:{leader:"ip_address:port"}}
  - 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.

2.2 Order Service Implementation:

2.2.1 Order Service initialization

- The port number that the current replica will be hosted on, the replica ID of the current replica, the ip address of the front end service, the port of the front end service, and a number denoting if caching should be utilized are passed into the order service as environmental variables at start time.

```
#Assign port number to this order service replica
port = sys.argv[1]
#Assign id number to this order service replica
replicaID = sys.argv[2]
#Assign front end server ip
frontendAdd = sys.argv[3]
#Assign front end server port
frontendPort = sys.argv[4]
#Determine if cache should be off/on 0/1 respectively.
cache = int(sys.argv[5])
```

- 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. Database files are written in form of ”order+replica_number.txt” e.g. ”order1.txt”, ”order2.txt”, etc.

```
#Check if database file exists in current folder, if it does load the json to a dict
if os.path.exists('./order'+str(replicaID)+'.txt'):
    with open('./order'+str(replicaID)+'.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
```

- Lock and empty dict of followers are initialized:

```
#Create Lock
lock = Lock()
#List of the current replicas followers. Will only be populated when the current replica is the leader
followers = {}

```

- On startup, a order server is initialized with a thread pool executor with a pool size of 10. This yields a thread per request architecture.

```
def serve():
    global transactions
    global transactionNum
    #Implement server with 5 threads
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    stockbazaar_pb2_grpc.add_OrderServicer_to_server(
        OrderServicer(), server)
    print("Order server started on: " + socket.gethostbyname(socket.gethostname())+':'+str(port))
    server.add_insecure_port(socket.gethostbyname(socket.gethostname())+':'+str(port))
```

2.2.2 Rejoined Replica Synchronization Technique

- When any leader election occurs within the front end service, (occurs at startup, occurs if the leader is unresponsive during POST requests, occurs if the leader is unresponsive during GET /orders/ requests.) the front end service calls the AssignFollowers() function of the designated leader replica. The AssignFollowers() function simply populates the current replicas list of followers (which starts empty on initialization) with the "ip_address:port" of its followers. (Can have 2, 1, or 0 followers) At any point, only the leader will have a populated followers list, since if the leader crashes and rejoins, it will be initialized to empty at start time.

```
    #Front end sends a list of the addresses of the follower replicas
def AssignFollowers(self, request, context):
    global followers
    #Check if first follower exists. If so, add to list of followers
    if len(request.followerOne) > 0:
        followers['followerOne'] = {'ip':request.followerOne.split(':')[0],
                                    'port':request.followerOne.split(':')[1]}
    #Check if second follower exists. If so, add to list of followers
    if len(request.followerTwo) > 0:
        followers['followerTwo'] = {'ip':request.followerTwo.split(':')[0],
                                    'port':request.followerTwo.split(':')[1]}
    print("Followers Assigned: " + str(followers))
    return stockbazaar_pb2.followersResult(resultFollowers = 1)
```

- At server start time, the order service does the following before serving any requests:
  1. Start the server.
  2. TRY to ping the front end service. If the front end service cannot be reached, CONTINUE. (This is part of the replica rejoining protocol. In typical operation of the application, the order services are started before the front end service. As such, the following is only called when the front end service is already started, which would entail that this current order service is rejoining.)
  3. Send a "GET /leader/" request to the front end service. This returns the "ip_address:port" of the current order service leader. (The current order service leader is always tracked and maintained by the front end service)
  4. Create a stub and connect to the order service leader replica.
  5. Pass the latest order number and address of the rejoining replica to the leader with the Synchronize function.
  6. Utilizing server streaming RPC, the leader replica continuously passes all the transactions that the rejoining replica has missed, while on each pass the rejoining replica propagates the transaction information to its in memory storage, as well as its physical database file.

```
    server.start()
    #Try to ping the front end service. In normal operating conditions, the front end service should not
    #be able to be pinged. If a replica is being restarted, then the front end service will already be
    #running and can be pinged.
    try:
        #Gets leader information from the frontend service.
        #Server-push technique to get the leader replica from the front end service
        conn = http.client.HTTPConnection(frontendAdd, frontendPort)
        conn.request('GET','/leader/' + socket.gethostbyname(socket.gethostname())+':'+str(port))
        rsp = conn.getresponse()
        data_received = rsp.read()
        data_received = json.loads(data_received)
        leaderAdd = data_received['data']['leader']
        conn.close()
        #Extract the ip address and port of the leader order service
        leaderIP, leaderPort = leaderAdd.split(':')[0], leaderAdd.split(':')[1]
        #Connect to the leader order service
        with grpc.insecure_channel(leaderIP + ':'+str(leaderPort)) as channel:
            stub = stockbazaar_pb2_grpc.OrderStub(channel)
            #Get the address of the replica that needs synchronization
            synchronizedAdd = socket.gethostbyname(socket.gethostname())+':'+str(port)
            #Pass the latest order number and address of the rejoining replica to the leader.
            for transaction in stub.Synchronize(stockbazaar_pb2.synchronizeTransNum(orderNum =
                                                transactionNum, synchronizedAdd = synchronizedAdd)):
                #propagate data into local memory
                transactionNum = transaction.orderNum
                transactions[transactionNum]={'name':transaction.stockName,'type':transaction.tradeType,
                                                        'quantity':transaction.stockTradeQuantity}
                print("Synchronized on Replica " + str(replicaID) + ": "
                      +str({transactionNum:transactions[transactionNum]}))
                #propagate data into the respective database file
                with open('./order'+str(replicaID)+'.txt', 'w') as file:
                     file.write(json.dumps(transactions)) # use `json.loads` to do the reverse
    except ConnectionRefusedError:
        pass
    server.wait_for_termination()
```

- The Synchronize function is defined as follows:
  1. Take the passed address of the replica that needs synchronization, and add to list of followers.
  2. Iterate through every transaction number in the database, and if the transaction number is larger than the latest transaction number of the replica that needs synchronization, pass the transaction information back to the rejoining replica. Continue until we reach latest transaction number of the leader replica. Iterative return is possible through RPC server-streaming.

```
    #Synchronize the transaction list of the leader with the rejoining replica
def Synchronize(self, request, context):
    global transactions
    global transactionNum
    global followers
    print("Replica " + str(replicaID) + " Synchornizing with " + str(request.synchronizedAdd))
    print("Followers Before Synchronization: " + str(followers))
    #Add replica to be synchronzied to the list of the leaders followers
    if len(followers.get('followerOne', {})) == 0:
        followers['followerOne'] = {'ip':request.synchronizedAdd.split(':')[0],
                                    'port':request.synchronizedAdd.split(':')[1]}
    elif len(followers.get('followerTwo', {})) == 0:
        followers['followerTwo'] = {'ip':request.synchronizedAdd.split(':')[0],
                                    'port':request.synchronizedAdd.split(':')[1]}
    print("Followers After Synchronization: " + str(followers))
    orderNeedsSynchronize = request.orderNum
    #Iterate through transactions s.t. we stream all order numbers
    #(rejoin replica max order num, leader max order num]
    for orderNum in list(transactions):
        if int(orderNum) > orderNeedsSynchronize:
            yield stockbazaar_pb2.transNumInfo(orderNum = int(orderNum),
                                               stockName = transactions[orderNum]['name'],
                                               tradeType = transactions[orderNum]['type'],
                                               stockTradeQuantity = transactions[orderNum]['quantity'])
```

2.2.3 Order Service on Trade Request

- All error handling associated with a trade request is handled on the order service. On a trade request, the leader order service replica 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)

```python
#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
    global followers
    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)
```

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)
8. Do the following with a lock (locking is required for the following steps because without locking, sometimes clients are given the same transaction number for different transactions):
   (a) If caching is turned on, commit a server-push invalidation request by passing the ”GET /cache/<stock name>” HTTP request to the front end service. (On a successful trade, we need to tell the front end service that its in memory cache is now stale. i.e. invalidation request)
   (b) Iterate the transaction number.
   (c) Map the transaction number to a dict containing the name of the stock traded, the trade type, and the quantity of stock traded.
   (d) To begin follower propagation we iterate through the list of followers as follows:
      i. Try to connect to the followers address. (Accounts for follower death.) If the follower cannot be reached, delete it from the list of followers.
      ii. Create a stub and connect to the given follower replica.
      iii. With the created stub, call the Propagate function.
   (e) Write the current in memory dict/database as a json to a txt file.
   (f) return the current transaction number.

```python
        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))

            with lock: #W/o locking, sometimes different clients are given the same transaction number
                if cache == 1:
                    #Server-push technique to tell frontend service to delete the stock from
                    #its local cache
                    conn = http.client.HTTPConnection(frontendAdd, frontendPort)
                    conn.request('GET','/cache/' + stockName)
                    rsp = conn.getresponse()
                    data_received = rsp.read()
                    conn.close()

                transactionNum += 1
                #Write to database in memory
                transactions[transactionNum]={'name':stockName,'type':tradeType,
                                              'quantity':stockTradeQuantity}
                print("Written on Replica " + str(replicaID) + ": "+
                      str({transactionNum:transactions[transactionNum]}))

                #propagate to all of the followers (given the current replica is the leader,
                #since if it was not the leader than this function would never even be called)
                for followerNum in list(followers):
                    followerAdd = followers[followerNum]
                    #First check to see if the given addres and port even has an existing
                    #gRPC server
                    try:
                        #If the follower can be connected to, propagate the information
                        replicaChannel = grpc.insecure_channel(str(followerAdd['ip']) + ':'
                                                               +str(followerAdd['port']))
                        stub = stockbazaar_pb2_grpc.OrderStub(replicaChannel)
                        res = stub.Propagate(stockbazaar_pb2.propagateOrderNum(orderNum =
                                                                       transactionNum,
                                                                       stockName = stockName,
                                                                       tradeType=tradeType,
                                                                       stockTradeQuantity =
                                                                       stockTradeQuantity))
                    except grpc._channel._InactiveRpcError:
                        print("Follower address does not exist "+str(followerAdd['ip'])+":"+
                              str(followerAdd['port']))
                        #If we cannot connect to a follower, delete it from the follower list
                        #(We assume that is has crashed)
                        del followers[followerID]

                #Write the current order database to a txt file in json format
                with open('./order'+str(replicaID)+'.txt', 'w') as file:
                     file.write(json.dumps(transactions)) # use `json.loads` to do the reverse

                return stockbazaar_pb2.requestResult(transactionNum = transactionNum)
```

- The Propagate function is defined as follows:
  1. Iterate the transaction number of the current replica.
  2. Propagate the passed data into local memory.
  3. Propagate the passed data into the physical database.

```
def Propagate(self, request, context):
   global transactions
   global transactionNum
   #propagate increase in transaction number
   transactionNum += 1
   #propagate data into local memory
   transactions[transactionNum]={'name':request.stockName,'type':request.tradeType,
                                 'quantity':request.stockTradeQuantity}
   #propagate data into the respective database file
   with open('./order'+str(replicaID)+'.txt', 'w') as file:
        file.write(json.dumps(transactions)) # use `json.loads` to do the reverse
```

```python
    print("Propagated on Replica " + str(replicaID) + ": "+str({transactionNum:transactions[transactionNum]}))
    return stockbazaar_pb2.propagateResult(resultPropagate = 1)
```

2.2.4 Order Service on Order Query Request

- On an order query, the leader order service replica does the following:
    1. Get the dict associated with the current transaction number from the in memory dict/database.
    2. If the length of the dict returned is 0, we know that the inputted transaction number does not exist. return (transaction number = -1, stock name = "", trade type = "", trade quantity = -1) to denote that the stock was not found.
    3. Else return (transaction number, stock name, trade type, trade quantity) as read from the dict returned from step 1

```python
#Takes the int orderNum and returns the order number, name of stock, type of trade, and quantity of stock traded.
def Lookup(self, request, context):
    global transactions
    #Get stock price for the given stock name
    stockInfo = transactions.get(request.orderNum, {})
    #If the name cannot be found in dict, return -1 as order num
    if len(stockInfo) == 0:
        return stockbazaar_pb2.lookupOrderNumResult(orderNum = -1, stockName = "", tradeType = "",
                                                    stockTradeQuantity = -1)
    #If found, return the order number, name of stock, type of trade, and quantity of stock traded.
    else:
        return stockbazaar_pb2.lookupOrderNumResult(orderNum = request.orderNum,
                                                    stockName = stockInfo['name'],
                                                    tradeType = stockInfo['type'],
                                                    stockTradeQuantity = stockInfo['quantity'])
```

2.3 Catalog Service Implementation:

2.3.1 Catalog Service Initialization

- 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':100,'quantity':1000}
    ford = {'price':11.93,'volume':100,'quantity':1000}
    apple = {'price':152.59,'volume':100,'quantity':1000}
    amazon = {'price':94.88,'volume':100,'quantity':1000}
    nvidia = {'price':240.63,'volume':100,'quantity':1000}
    intel = {'price':28.01,'volume':100,'quantity':1000}
    meta = {'price':194.02,'volume':100,'quantity':1000}
    nike = {'price':126.13,'volume':100,'quantity':1000}
    amc = {'price':4.78,'volume':100,'quantity':1000}
    gamestop = {'price':20.00,'volume':100,'quantity':1000}

    stockData['apple']=apple
    stockData['tesla']=tesla
    stockData['ford']=ford
    stockData['amazon']=amazon
    stockData['nvidia']=nvidia
    stockData['intel']=intel
    stockData['meta']=meta
    stockData['nike']=nike
    stockData['amc']=amc
    stockData['gamestop']=gamestop

    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 10. This yields a thread per request architecture. Locking is not utilized since threadpoolexecutor utilizes GIL.

2.3.2 Catalog Service on Lookup Request

- 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)
```

```python
            #If found, return stock price and stock volume
            else:
                return stockbazaar_pb2.lookupResult(stockPrice=stockInfo['price'],stockQuantity=stockInfo['quar
```

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.

```python
         #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)
```

2.4 Front End Implementation:

2.4.1 Front End Initialization

- The replica ID, replica address, replica ID, replica address, replica ID, replica address, and a number denoting if caching should be turned on or off are passed as environmental variables to the front end service at start time. Replicas are held in a dict mapping replica ID to its respective address. (Addresses are in the form ”ip_address:port”) Lock is initalized here as well.

```
#Read ID numbers and address of replica 1
oneOrderID = sys.argv[1]
oneOrderAdd = sys.argv[2]
#Read ID numbers and address of replica 2
twoOrderID = sys.argv[3]
twoOrderAdd = sys.argv[4]
#Read ID numbers and address of replica 3
threeOrderID = sys.argv[5]
threeOrderAdd = sys.argv[6]
#Determine if cache should be off/on 0/1 respectively.
cache = int(sys.argv[7])

#Map replicas to a dict
replicas = {oneOrderID:oneOrderAdd,twoOrderID:twoOrderAdd,threeOrderID:threeOrderAdd}
#Create Lock
lock = Lock()
```

- 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.

```
#Create pool mix in that allows for requests to be submitted to pool.
class PoolMixIn(ThreadingMixIn):
    def process_request(self, request, client_address):
        self.pool.submit(self.process_request_thread, request, client_address)

def run():
    print('http server is starting...')
    print("Front-end server started on: " + socket.gethostbyname(socket.gethostname())+':56893')
    #Define the pooled http server.
    class PoolHTTPServer(PoolMixIn, HTTPServer):
        pool = ThreadPoolExecutor(max_workers=10)

    server = PoolHTTPServer((socket.gethostbyname(socket.gethostname()), 56893), StockHTTPRequestHandler)
    print('http server is running...')
    server.serve_forever()

if __name__=="__main__":
    cataloghost = os.getenv("CATALOG_HOST", socket.gethostbyname(socket.gethostname()))
    orderhost = os.getenv("ORDER_HOST", socket.gethostbyname(socket.gethostname()))
    #Create local cache
    cachedStockData = {}
    #Perform leader election for order service replicas
    currLeaderID = -1
    currLeaderID, orderhost, orderport = leaderElection(replicas, currLeaderID)
    run()
```

- Empty dict that will contain the local cache is initialized here.
- The variable denoting which replica ID is the current leader is initialized to -1 here.

2.4.2 Leader Election and Pinging

- Leader election is performed in the following situations:
  1. Leader election is performed at front end start time.
  2. Leader election is performed when the leader is unresponsive during a POST request.
  3. Leader election is performed when the leader is unresponsive during a GET /orders/ request.
- Leader election is performed as follows:
  1. Sort the list of replicas by their replica ID in descending order.
  2. Iterate through the list of sorted replica IDs.
     (a) If the replica ID is equal to the current leader's ID, skip this iteration. (We assume that leaderElection() is only called when there is a leader failure. On start up, the current leader's ID is initialized to -1 so we will not skip any replicas during leader candidacy.)
     (b) With the current replica ID, get the respective IP address and port number.
     (c) Attempt to ping the IP address and port number from step b.
     (d) If able to be successfully pinged, create a stub and connect to the order service defined by the ip address and port number from step b.
     (e) If the replica ID is the first number in the sorted list, we know that it will have two followers.
     (f) If the replica ID is the second largest number in the sorted list, we know that it will have one follower. (Cases where the largest replica ID cannot be reached, or if the current leader is the largest replica ID and we assume that it has crashed.)
     (g) If the replica ID is the third largest number in the sorted list, we know that it will have no followers. (Cases where the largest and second largest replica ID cannot be reached, or if the current leader is the second largest replica ID and we assume that it has crashed.)
     (h) With the stub we call the AssignFollowers() function of the elected leader replica, and pass the followers to it.
     (i) return the ID of the newly elected leader, the IP address of the newly elected leader, and the port of the newly elected leader.

```python
def leaderElection(replicas, currLeaderID):
    #Sort replica ID numbers from largest to smallest
    replicaIDs = list(replicas.keys())
    replicaIDs.sort(reverse=True)
    #Iterate down through list of replica ID numbers
    for x in range(0,len(replicaIDs)):
        leader = replicaIDs[x]
        #If currLeaderID=-1, which means a leader has not been elected, then the if statement will
        #never be called
        if leaderCandidate == currLeaderID:
            continue
        else:
            leaderCandidateIP = replicas[leaderCandidate].split(':')[0]
            leaderCandidatePort = replicas[leaderCandidate].split(':')[1]
            #If leader is able to be succesfully pinged
            if ping(leaderCandidateIP, leaderCandidatePort):
                #Once we have determined who the leader is, we assign their followers by sending their
                #followers in the following format "ip_address:port"
                with grpc.insecure_channel(leaderCandidateIP + ':' +str(leaderCandidatePort)) as channel:
                    stub = stockbazaar_pb2_grpc.OrderStub(channel)
                    #If the leader is the largest ID, then the two smaller ID's will be the followers
                    if x == 0:
                        followerOne, followerTwo = replicas[replicaIDs[x+1]], replicas[replicaIDs[x+2]]
                    #If the leader is the second largest ID, the the smallest ID will be the follower.
                    #(We assume that the second largest ID will only ever be the leader when the
                    #largest ID cannot be reached)
                    elif x == 1:
                        followerOne, followerTwo = replicas[replicaIDs[x+1]], ""
                    #If the leader is the third largest ID, we assume that both the largest and
                    #second largest IDs cannot be reached and as such no followers
                    else:
                        followerOne, followerTwo = "", ""
                    res = stub.AssignFollowers(stockbazaar_pb2.followersAssigned(followerOne = followerOne,
                                                                                 followerTwo = followerTwo))
                return leaderCandidate, leaderCandidateIP, leaderCandidatePort
```

- The ping() function is defined as follows:
  1. We first attempt to connect to the given IP and port to see if a server at that address even exists.
  2. If the server exists, we attempt to see if the server is responsive, with a 5 second timeout.
  3. If any of these attempts fail, we return FALSE.
  4. else, we return TRUE

```
def ping(ipAdd,port):
    #First check to see if the given addres and port even has an existing gRPC server
    try:
        channel = grpc.insecure_channel(str(ipAdd) + ':'+str(port))
        #If the server exists at the address, check if the server is responsive
        try:
            grpc.channel_ready_future(channel).result(timeout=5)
            channel.close()
            return True
        except grpc.FutureTimeoutError:
            print(" Connection to "+str(ipAdd)+":"+str(port) + " timed out.")
            channel.close()
            return False
    except grpc._channel._InactiveRpcError:
        print(" Address does not exist "+str(ipAdd)+":"+str(port))
        return False
```

2.4.3 Front End on GET Request

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

- Extract the request type.

```
def do_GET(self):
    global orderhost
    global orderport
    global currLeaderID
    global cachedStockData
    #Extract GET request type i.e. /REQUEST/<input> we extract "REQUEST"
    type = self.path.split('/')[1]
```

- **If the request type is ”stocks” (GET /stocks/<stock name>)**
  1. Check the local cache to see if the stock exists. If it does, create a data dict containing {data:{name,price,quantity}}. Send a response code of 200.
  2. else, create a stub and connect to the catalog service. (port 56892)
  3. Call the lookup function of the catalog service.
  4. 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.
  5. else, create a data dict containing {data:{name,price,quantity}}. Send a response code of 200.
  6. If caching is turned on, add the retrieved data to the local cache.

```
    if type == 'stocks':
        #Extract the name of the stock from the HTTP request
        stockName = self.path.split('/')[2]
        #First check local cache to see if the stock is present
        if len(cachedStockData.get(stockName, {})) > 0:
            cachedStock = cachedStockData.get(stockName, {})
            #send code 200 response
            self.send_response(200)
            response = {}
            data = {'name':stockName,'price':cachedStock['price'],
                    'quantity':cachedStock['quantity']}
            response['data']=data
        #If the stock is not located in the cache, query the catalog service
        else:
            with grpc.insecure_channel(cataloghost + ':56892') as channel:
                stub = stockbazaar_pb2_grpc.CatalogStub(channel)
                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}
                    if cache == 1:
                        #Add the retrieved data to the local cache
                        cachedStockData[stockName] = {'price':round(res.stockPrice,2),
                                                      'quantity':res.stockQuantity}
                    response['data']=data
```

- **If the request type is "orders" (GET /orders/<transaction number>)**

  1. With a lock, we do the following: (Locking is required, since if the leader is unresponsive, multiple clients might trigger multiple leader elections. With locking, only one leader election can occur at a time)
  2. Attempt to connect to the leader. If we cannot connect, call the leaderElection() function. Proceed with the "GET /orders/<transaction number>" request on the newly elected leader.
  3. Create a stub and connect to the current leader replica's IP address and port.
  4. With the stub, call the lookup function for the given transaction number.
  5. If the returned transaction number is -1, we know that the given transaction number doesn't exist in the database. Create an error dict {error:{'code':404,'message':"transaction not found"}} when stock with given name not found. Send a response code of 404.
  6. else, create a data dict containing {data:{number,name,type,quantity}}. Send a response code of 200

```
    elif type == 'orders':
        #Extract the order num of the stock from the HTTP request
        orderNum = self.path.split('/')[2]
        orderNum = int(orderNum)
        def getorders():
            with grpc.insecure_channel(orderhost + ':' + str(orderport)) as channel:
                stub = stockbazaar_pb2_grpc.OrderStub(channel)
                res = stub.Lookup(stockbazaar_pb2.lookupOrderNum(orderNum = orderNum))
                if res.orderNum == -1:
                    #send code 404 response
                    self.send_response(404)
                    response = {}
                    error = {'code':404,'message':"transaction not found"}
                    response['error']=error
                else:
                    #send code 200 response
                    self.send_response(200)
                    response = {}
                    data = {'number':res.orderNum,'name':res.stockName,'type':res.tradeType,
                            'quantity':res.stockTradeQuantity}
                    response['data']=data
            return response

        with lock:
            try:
                response = getorders()
            #If the current order service replica crashes/timesout, elect a new leader, and execute
            #the GET request with a connection to the new leader
            except grpc._channel._InactiveRpcError:
                print("OLD LEADER REMOVED DURING GET /ORDERS/: " + str(currLeaderID), end = '')
                currLeaderID, orderhost, orderport = leaderElection(replicas, currLeaderID)
                print("/NEW LEADER ELECTED DURING GET /ORDERS/: " + str(currLeaderID))
                response = getorders()
```

- **If the request type is "cache" (GET /cache/<stock name>)**
  1. If caching is turned on, pop the dict key with the given stock name from the local cache.

```
         def do_GET(self):
    elif type == 'cache':
        #Extract the name of the stock from the HTTP request
        stockName = self.path.split('/')[2]
        if cache == 1:
            #On successful trade, we must remove the stock with given stock name from the local cache
            cachedStockData.pop(stockName, None)
        #send code 200 response
        self.send_response(200)
        response = {}
        data = {}
        response['data']=data
```

- **If the request type is "leader" (GET /leader/)**
  1. Create a data dict containing {data:{leader:"ip_address:port"}}

```
    elif type == 'leader':
        #send code 200 response
        self.send_response(200)
        response = {}
        data = {'leader':replicas[currLeaderID]}
        response['data']=data
```

- Send a header with "Connection: keep-alive" and "Content-length: size of dict"
- Convert the error/data dict to a json, encode the json, and write response.

```
    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
```

2.4.4 Front End on POST Request

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

1. With a lock, we do the following: Locking is required, since if the leader is unresponsive, multiple clients might trigger multiple leader elections. With locking, only one leader election can occur at a time)
2. Attempt to connect to the leader. If we cannot connect, call the leaderElection() function. Proceed with the "POST /orders" request on the newly elected leader.
3. Create a stub and connect to the leader replica order service.
4. 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.
5. 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.
6. 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.
7. If the returned transaction number = -2, create an error dict {error:{'code':400,'message':"invalid request type"}} Send a response code of 400.
8. If the returned transaction number = -3, create an error dict {error:{'code':404,'message':"invalid number of stocks"}} Send a response code of 400.
9. 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.
10. If no error is returned, create a data dict containing {data:{transaction_num}}. Send a response code of 200.
11. If no error is returned, create a data dict containing {data:{transaction_num}}. Send a response code of 200.
12. Send a header with "Connection: keep-alive" and "Content-length: size of dict"
13. Convert the error/data dict to a json, encode the json, and write response.

```
def do_POST(self):
    global orderhost
    global orderport
    global currLeaderID

    #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']

    def post():
        with grpc.insecure_channel(orderhost + ':' + str(orderport)) as channel:
            stub = stockbazaar_pb2_grpc.OrderStub(channel)
            res = stub.Request(stockbazaar_pb2.requestStockName(stockName = stockName,
                                stockTradeQuantity = inputData['quantity'],
                                tradeType = inputData['type']))
            res = res.transactionNum
            #Stock not found
```

```python
            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
        return response

    with lock:
        try:
            response = post()
        #If the current order service replica crashes/timesout, elect a new leader, and execute the POST
        #request with a connection to the new leader
        except grpc._channel._InactiveRpcError:
            print("OLD LEADER REMOVED DURING POST: " + str(currLeaderID), end = '')
            currLeaderID, orderhost, orderport = leaderElection(replicas, currLeaderID)
            print("/NEW LEADER ELECTED DURING POST: " + str(currLeaderID))
            response = post()

    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
```

2.5 Client Implementation:

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

  1. Select a stock name from: [”tesla”,”ford”,”apple”,”amazon”,”nvidia”,”intel”,”meta”,”nike”,”amc”,”gamestop”,”imagi company”] which represents 10 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.
     (d) If the trade is returned as successful, add the stock information to the local cache. Mapping {transaction number:{transaction number, name, type, quantity}}.

```python
#Locally store order information for successful trades in order to verify with backend
localOrderInfo = {}

#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 
mode = int(sys.argv[4]) #0 for verbose 1 for non-verbose/latency and verification shows
#create a connection
conn = http.client.HTTPConnection(http_server, port)
stockNames = ["tesla","ford","apple","amazon","nvidia","intel","meta","nike","amc","gamestop"] #stockNames = ["tesl


#Counts of times for each request
catalogLookup = 0.0
orderLookup = 0.0
orderTrade = 0.0
#Running total of number of requests sent s.t. we can calculate average latency
numCatalogLookup = 0
numOrderLookup = 0
numOrderTrade = 0

for _ in range(100):
    stockName = random.choice(stockNames)
    if mode == 0: print("-------------- Lookup --------------")
    if mode == 0: print("Input: " + stockName)
    start_time = timeit.default_timer()
    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)
    catalogLookup += (timeit.default_timer() - start_time)
    numCatalogLookup += 1
    if mode == 0: print("Output: " + str(data_received))

    order = random.uniform(0,1)
    if probability > order:

        if mode == 0: print("-------------- Trade --------------")
        data = {}
        data['name']=stockName
        #Randomly choose to buy or sell or invalid option
        tradeTypes = ["buy", "sell"] #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(1,100) #stockQuantity = random.randint(-10,100)
        data['quantity']=stockQuantity
        json_data = json.dumps(data)
        if mode == 0: print("Input: " + str(json_data))
        start_time = timeit.default_timer()
        conn.request('POST', '/orders', body = json_data.encode(), headers={'Content-Length':str(len(bytes(json_dat
        #get response from server
        rsp = conn.getresponse()
        data_received = rsp.read()
        #Convert data to dict
        data_received = json.loads(data_received)
        orderTrade += (timeit.default_timer() - start_time)
        numOrderTrade += 1

        #Verify that the trade was successful s.t. we can add it to our local record
        if(len(data_received.get('error',{})) == 0):
            orderNum = (data_received['data'])['transaction_number']
            orderInfo = {'number':orderNum,'name':stockName,'type':tradeType,'quantity':stockQuantity}
            localOrderInfo[orderNum] = orderInfo
        if mode == 0: print("Output: " + str(data_received))
    if mode == 0: print("\n")

```

2.5.1 Client Order Query Verification

- Iterate through the local cache, and for each transaction number, send a ”GET /orders/<transaction number>” request to the front end service. If at any point, a transaction does not match the transaction returned from the front end, print ’A NON MATCH WAS FOUND’

```
#Perform order verification comparing local info to server info
print("----- Order Query Verification -----")
print("Successful Trades Made: " + str(len(localOrderInfo)))
count = 0
for orderNum, orderInfo in localOrderInfo.items():
    if mode == 0: print("------- Verifying Order " + str(orderNum) + " -------")
    start_time = timeit.default_timer()
    conn.request('GET','/orders/' + str(orderNum))
    #get response from server
    rsp = conn.getresponse()
    data_received = rsp.read()
    #Convert data to dict
    data_received = json.loads(data_received)
    orderLookup += (timeit.default_timer() - start_time)
    numOrderLookup += 1
    if mode == 0: print("Local: " + str(orderInfo))
    if mode == 0: print("Server: " + str(data_received['data']))
    #Verify that the two dicts match
    if orderInfo == data_received['data']:
        if mode == 0: print("MATCH")
        pass
    else:
        if mode == 0: print("NO MATCH")
        pass
        count += 1
    if mode == 0: print()
if count != 0:
    print("A NON-MATCH WAS FOUND")
else:
    print("ALL ORDERS VERIFIED")
```

```python
print("---- Average Latency of Requests ----")
print("Average Latency of Catalog Lookup Request: " + str(catalogLookup/numCatalogLookup))
if numOrderTrade == 0:
    print("Average Latency of Order Trade Request: N/A")
else:
    print("Average Latency of Order Trade Request: " + str(orderTrade/numOrderTrade))
if numOrderLookup == 0:
    print("Average Latency of Order Trade Request: N/A")
else:
    print("Average Latency of Order Query Lookup: " + str(orderLookup/numOrderLookup))

conn.close()
```

3 Macro-level Implementations:

3.1 System Design

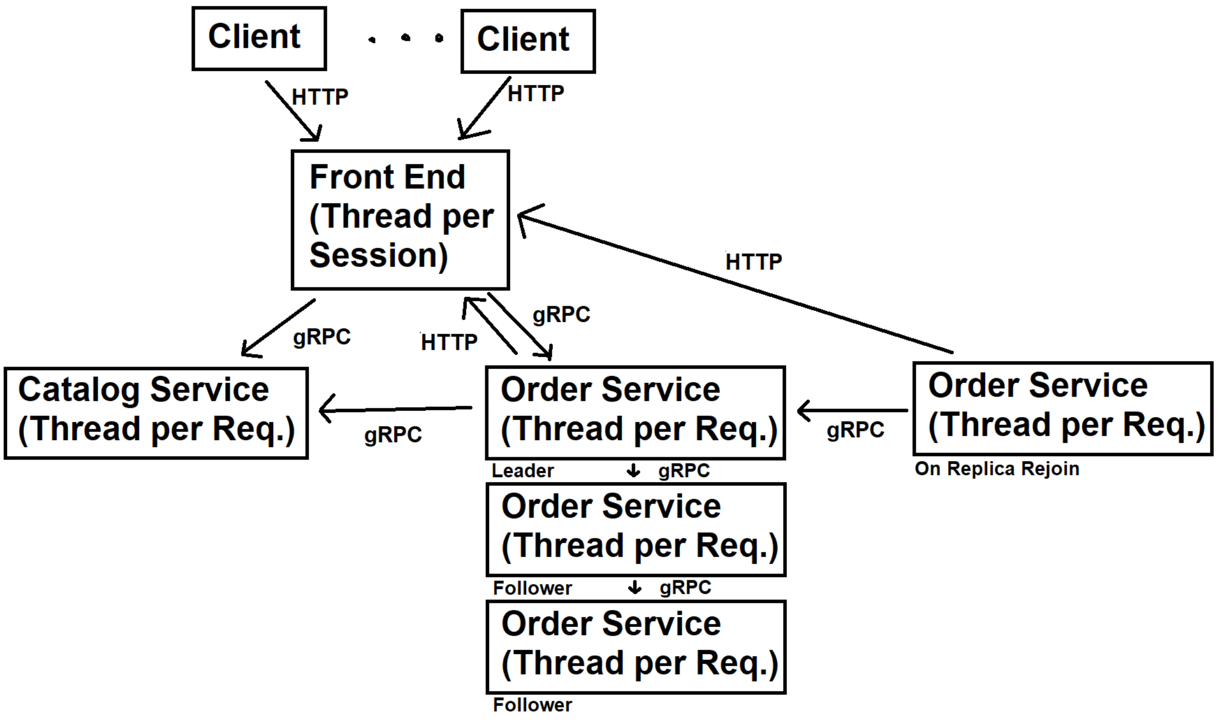
Client
Client
HTTP
HTTP
Front End (Thread per Session)
HTTP
gRPC
gRPC
HTTP
Catalog Service (Thread per Req.)
gRPC
Order Service (Thread per Req.)
Leader
gRPC
gRPC
Order Service (Thread per Req.)
On Replica Rejoin
Order Service (Thread per Req.)
Follower
gRPC
Order Service (Thread per Req.)
Follower

Figure 2: Diagram showing type of requests that are sent in the stock server architecture.

- The catalog service will always be on port 56892. The order service can be any 3 non-equal ports. 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.

3.2 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).

3.3 gRPC:

- gRPC is utilized to handle the following requests:
  - Front end to catalog service.
  - Front end to order service.
  - Order service to catalog service.
  - Order service to order service on propagation.
  - Order service to order service on rejoin/synchronization.

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

3.3.1 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.
  - Lookup
    - input: int containing the transaction number of the transaction.
    - returns: int containing the transaction number of the transaction. string containing the stock name. string containing the trade type. int containing the quantity of stock traded.
  - Propagate
    - input: int containing the transaction number of the transaction. string containing the stock name. string containing the trade type. int containing the quantity of stock traded.
    - returns: int containing the result of the propagation. (1 on success)
  - AssignFollowers
    - input: string containing the first follower. string containing the second follower.
    - returns: int containing the result of the assignment. (1 on success)
  - Synchronize
    - input: int containing the transaction number of the latest transaction of the rejoining replica. string containing the ”ip_address:port” of the rejoining replica.
    - returns: stream of the following: int containing the transaction number of the transaction. string containing the stock name. string containing the trade type. int containing the quantity of stock traded.