Asterix and Bazaar
Design Document

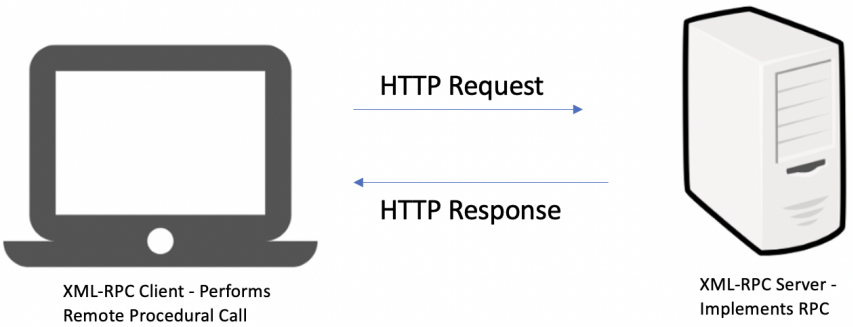
Group:

Prachiti Parkar
Sai Pranav Kurly

Working

We have used XML-RPC client-server architecture to simulate a P2P network along with multithreading so that all the servers and the functions can work concurrently.

XML-RPC - It is a remote procedural protocol which uses XML to encode its calls and HTTP as its transport mechanism.

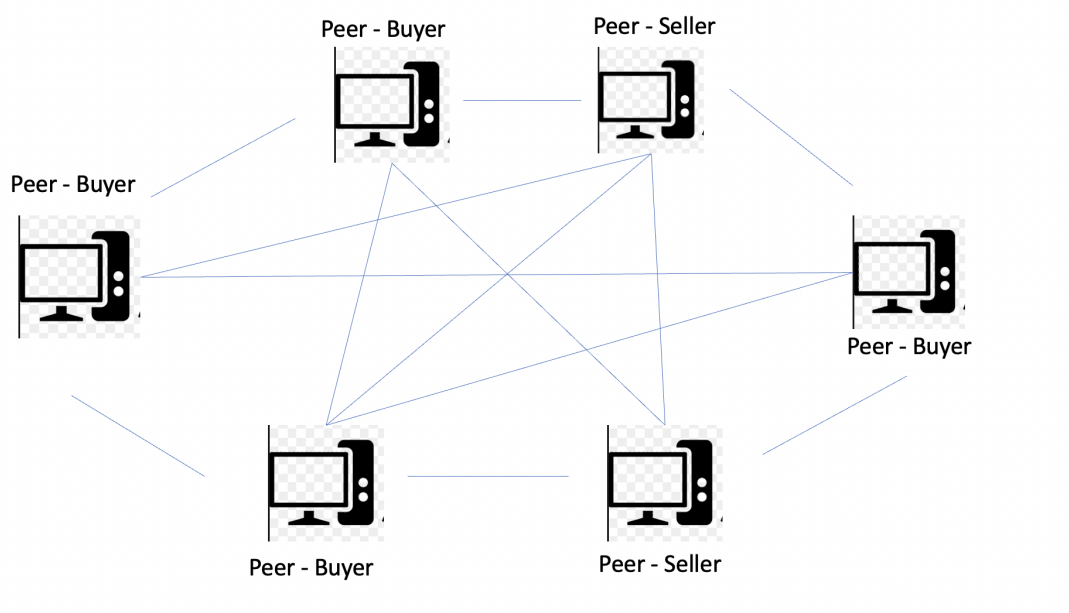


Why XML-RPC?

1. It is easy to implement using Python libraries while allowing complex structures to be transmitted, processed and returned.
2. It offers integrators an opportunity to use a standard vocabulary and approach for exchanging information.
3. It is excellent to connect various computers i.e in our case various peers on different ports.
4. XML-RPC uses the HTTP protocol to pass information from a client computer to a server computer. It can easily be tested locally and on remote systems

Peer to Peer Network

Every peer node (buyer or seller) behaves as a XML-RPC client as well as a XML-RPC Server. All the peers are initialized as XML-RPC servers and if a function call (buy, reply, lookup) needs to be made, the client performs a RPC call to the appropriate server.



A sample of P2P network with 4 buyers and 2 sellers

Deep Dive:

Every Peer is a thread and all peers run concurrently to form a multithreaded system. Thereafter, each peer initializes its own server. We are running our code on localhost and hence, every peer will be run on localhost with a different port number. The port number will be 49151 [usually the unregistered port] + peerId. Each server registers the functions which are appropriate as per the role and starts the server which serves continuously until a keyboard interrupt. When a peer has to make a RPC call, it serves as a XML-RPC client and makes a call to the respective XML-RPC server.

Functions registered as per the role:

1. getCurrentTime()
   Registered with both buyers and sellers, this is to make sure that the peer server is up and running before making any further RPC calls.

2. lookup()
   Registered with both buyers and sellers, every buyer and seller can perform a lookup to find a seller for the appropriate buyer. This call is started by the buyer who is looking for a seller who has the product it needs. It traverses through hops until a max hop count (defined beforehand) is reached. If the max hop count is reached, the search for the product stops.

3. buy()
   Registered with only sellers, this results in decrementing the units of a particular product sold by the seller if the seller hasn't started selling another product or if there are no units of the product left. This is not registered with the buyer since the buyer isn't selling any product for which the quantity needs to be decremented.

4. reply()

Registered with both sellers and buyers. Once an appropriate seller is found for a buyer, the reply request is again sent back from the seller to the buyer through the same path it reached the seller. Hence, the intermediaries can be either a buyer or seller.

Code Walkthrough:

Roles: We have set the first peer as buyer and second as seller since there needs to be 1 buyer and 1 seller in the marketplace. For the remaining peers, the roles are randomized.

Neighbors: We have used an adjacency matrix to store neighbors of peers. This is easier as well as efficient as we don't spend much time searching for the neighbor which would have otherwise increased the response time.

Hop Count:

Thinking: The maximum distance possible between n peers is less than or equal to n-1. Thus, the hop count can range between [1,totalPeers-2].

Peers: For 2 peers, the maximum distance is 1, so the hop count = 0. This case is invalid. Hence, we don't consider 2 peers. The number of total peers which leads to valid Bazaar can range between 3 and 6 both included.

Shortest distance: We have used Dijikstra's algorithm to calculate the shortest distance between every 2 peers and the maximum among these distances is stored. Hop count is then randomly defined between [1, maximum distance - 1] during runtime. Since the hop count is randomly defined, we were able to derive its relationship with the performance metrics.

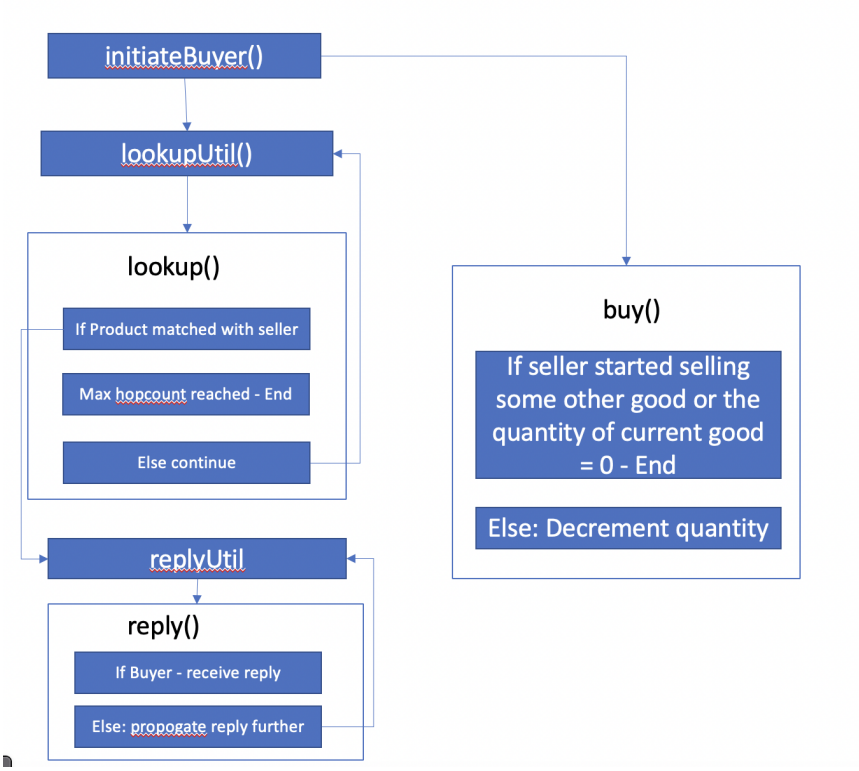
Client Wait Time: Defined as 4 (after trial and error, this was the client wait time which captured sellers appropriately for upto 6 peers)

Max Units (seller selling a product): Defined as 10

I. Buyer

Calls initiateBuyer() which in turn calls lookupUtil(); this is responsible for finding the peer's client proxy. It then calls the lookup() which looks for the seller. Once an appropriate seller is found, the seller sends a reply which propagates through the same path to reach the buyer.

a. The buyer waits till a client timeout time to get a list of all potential sellers, it chooses the first one which has the least response time and calls buy which decrements the quantity if appropriate conditions are met under lock (synchronization) so that no other buyer can buy the same product from the same seller at the same time.
b. If there is no seller selling the product which a buyer needs, then there will be no potential sellers. And after a client timeout time, the buyer will start buying a new product (May or may not be the same as the earlier product, it is randomized).



A sample of 1 buyer searching for its product

II. Seller

Calls initiateSeller(), finds a product to sell and waits for a buyer call and then sends a reply. (Similar to above diagram -> [If Product matched with seller]) If the buyer accepts a request, it calls buy() which lets the seller know that it's product has been bought, and it decrements the quantity of the product. If the quantity reaches 0, it starts selling a new product (May or may not be the same as the earlier product - randomized).

Design Tradeoffs

1. We made the choice of using XML-RPC architecture since it is easy to implement and design but it uses HTTP to make requests and send responses thus incorporating all the inefficiencies of HTTP. HTTP has limitations with respect to distributed computing requirements. Though it is easy to use, it might lead to poor performance.
2. Since we are using HTTP, it is not secure. The server's public key might not be registered with the CA i.e no SSL certificate thus, there is no guarantee that the transmission is secure. If we were also supposed to authenticate payments for Bazaar, the peer servers could have been a victim of the Man-In-The-Middle attack.

Improvements:

1. **Increase CPU utilization**: To make better use of the computational resources of multi-core machines, multiprocessing or concurrent.futures.ProcessPoolExecutor could have been used. This would have decreased latency as well as response time.
2. **Reduce thread overheads and decrease response time**: Instead of starting a new thread for each new task, we could have used a thread pool which could reuse previously created threads so that it minimizes thread overhead. Since a thread would have already

been created, it would reduce the time of thread creation and thus making the entire network more responsive.

3. **Use https for the hosts:** This would make our application secure which can handle payment requests if required in future.
4. **Reduce same seller replies:** It is possible that we have 2 same sellers in the potential sellers list through 2 different paths. Since we are choosing the seller with the lowest response time, it would be efficient if we dont send a lookup request to the same seller again. This would decrease the number of requests and the average response time.

How to Run

1. Go to the src folder inside the main folder.
   cd src
2. Command - python3 peer.py #numberOfPeer
   Eg:
   python3 peer.py 4 [Number of peers = 4]
   To change the neighbor mapping, the below dictionary can be changed. The dictionary key is the number of nodes passed as argument and value is the neighbor mapping.

```
#------------Change neighbor mapping here:----------------
#Initializing Graph for various nodes
nodeMapping = {
    3: [[False, True, True],
        [True, False,False],
        [True, False, False]],
    4: [[False, True, False, False],
        [True, False, True, True],
        [False, True, False, True],
        [True, True, True, False]],
    5: [[False, True, False, False, False],
        [True, False, True, False, False],
        [False, True, False, True, False],
        [False, False, True, False, True],
        [False, False, False, True, False]],
    6: [[False, True, False, False, False, False],
        [True, False, True, False, False, False],
        [False, True, False, True, False, False],
        [False, False, True, False, True, False],
        [False, False, False, True, False, True],
        [False, False, False, False, True, False]]
}
```

For Testing with customized roles and neighbor map:

1. Command - python3 test.py #testCaseNumber [testCases are mentioned in test.py]

Note: If adding a new test case, add the roles and neighbor map in testMapping to a new test case number and use that in the command.

test.py is the same as peer.py with test cases to thoroughly test the functionality of the application.

TestCases

To run the test cases the test.py file must be used. **The arg passed to the script is the test case number.**

1. If more than one peer is passed as input

```
Distributed Systems/Lab/Lab-1/Cloned/CS677/src$ python3 test.py 1
Enter more than 1 peer!
```

2. If there is a least one seller

```
Distributed Systems/Lab/Lab-1/Cloned/CS677/src$ python3 test.py 2
Atleast one seller must be present
```

3. If there is a least one buyer

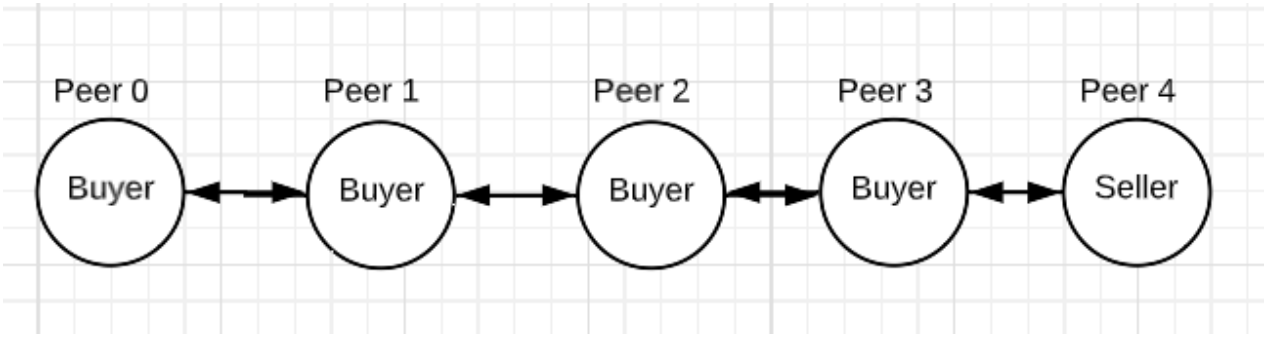
```
Distributed Systems/Lab/Lab-1/Cloned/CS677/src$ python3 test.py 3
Atleast one buyer must be present
```

4. If a peer has more than 3 neighbors

```
Distributed Systems/Lab/Lab-1/Cloned/CS677/src$ python3 test.py 4
Peer has more than 3 neighbors!
```

5. If request is discarded if number of hops is greater than the set limit

   Considered 5 peers and hop count limit 2



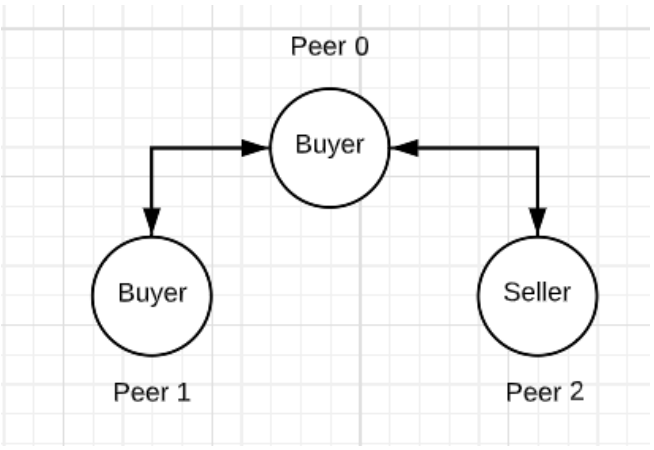
```
Distributed Systems/Lab/Lab-1/Cloned/CS677/src$ python3 test.py 5
Running on: localhost
Number of nodes: 5
Graph:
[False, True, False, False, False]
[True, False, True, False, False]
[False, True, False, True, False]
[False, False, True, False, True]
[False, False, False, True, False]
Marketplace is live! Check output.txt in PeerID directory to check the logging
```

Peer 1 should never reach node 4 because of the hop limit 2. Below we can see node 3's log and notice that the hop count has reached and lookup has been stopped and no reply has been sent back. The path 3-2-1 shows that the request originated from node one and ended at node 3.

src > Peer3 > output.txt

```
1  2022-10-18 16:16:24.607121 Max Hopcount Reached, Can't lookup futher. Lookup stopped Path: 3-2-1
2  2022-10-18 16:16:25.581423 Peer 3 plans to buy Salt
3  2022-10-18 16:16:25.627344 Lookup for product Fish propogated from peerID 3 to peerID 4
4  2022-10-18 16:16:25.665964 Peer 4 has Fish
5  2022-10-18 16:16:25.677059 Propogate the reply to peerID 2 Path: 2
6  2022-10-18 16:16:29.590480 Stopped buying Salt because no sellers
7  2022-10-18 16:16:32.604605 Peer 3 plans to buy Boar
8  2022-10-18 16:16:32.631016 Max Hopcount Reached, Can't lookup futher. Lookup stopped Path: 3-2-1
9  2022-10-18 16:16:34.620888 Lookup for product Salt propogated from peerID 3 to peerID 4
10 2022-10-18 16:16:36.611579 Stopped buying Boar because no sellers
11 2022-10-18 16:16:39.613367 Peer 3 plans to buy Fish
12 2022-10-18 16:16:39.672250 Peer 4 has Fish
13 2022-10-18 16:16:39.685764 Received a reply from peerID 4
14 2022-10-18 16:16:41.606760 Max Hopcount Reached, Can't lookup futher. Lookup stopped Path: 3-2-1
15 2022-10-18 16:16:43.640018 Lookup for product Fish propogated from peerID 3 to peerID 4
16 2022-10-18 16:16:43.662839 Hurry! PeerID 4 has Fish to sell
```

6. If sellers sells all good then pick another random good and start selling



Peer 1 (Seller):

output.txt ...\Peer0 M | output.txt ...\Peer1 M | peer.py M

src > Peer1 > output.txt

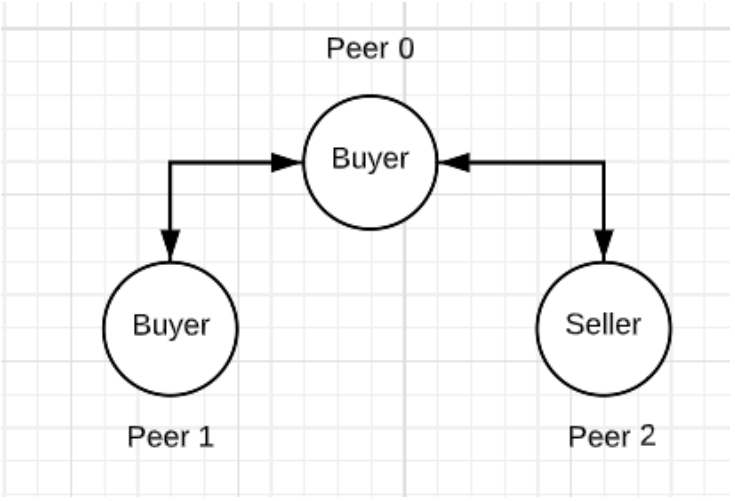
```
2022-10-17 18:31:26.461514 Selling Fish: 1 Unit(s)
2022-10-17 18:31:49.477553 Remaining Fish: 0 Unit(s)
2022-10-17 18:31:49.478914 Selling new good Boar: 6 Unit(s)
2022-10-17 18:32:27.526275 Remaining Boar: 5 Unit(s)
2022-10-17 18:32:41.543473 Remaining Boar: 4 Unit(s)
```

output.txt ...\Peer0 M | output.txt ...\Peer1 M | peer.py M | node.py

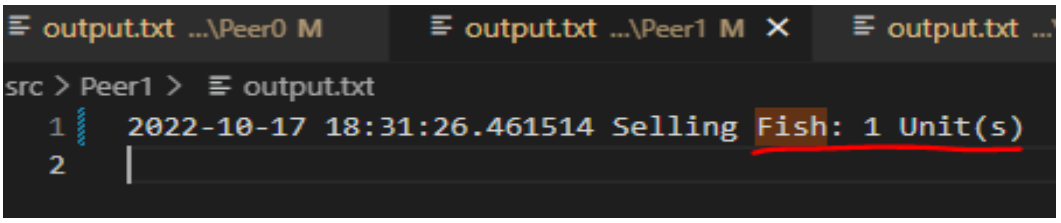
src > Peer0 > output.txt

```
2022-10-17 18:31:30.461399 Peer 0 plans to buy Boar
2022-10-17 18:31:34.462895 Stopped buying Boar because no sellers
2022-10-17 18:31:38.463802 Peer 0 plans to buy Salt
2022-10-17 18:31:42.465955 Stopped buying Salt because no sellers
2022-10-17 18:31:45.467549 Peer 0 plans to buy Fish
2022-10-17 18:31:45.474163 Peer 1 has Fish
2022-10-17 18:31:45.490030 Received a reply from peerID 1
2022-10-17 18:31:49.480126 Hurry! PeerID 1 has Boar to sell
2022-10-17 18:31:49.484574 Bought Fish from peerID 1
The average response time: 0.021311
2022-10-17 18:31:53.486835 Peer 0 plans to buy Fish
2022-10-17 18:31:57.488872 Stopped buying Fish because no sellers
2022-10-17 18:32:01.490647 Peer 0 plans to buy Fish
2022-10-17 18:32:05.493695 Stopped buying Fish because no sellers
2022-10-17 18:32:08.497109 Peer 0 plans to buy Salt
2022-10-17 18:32:12.499871 Stopped buying Salt because no sellers
2022-10-17 18:32:16.500768 Peer 0 plans to buy Salt
2022-10-17 18:32:20.502894 Stopped buying Salt because no sellers
```

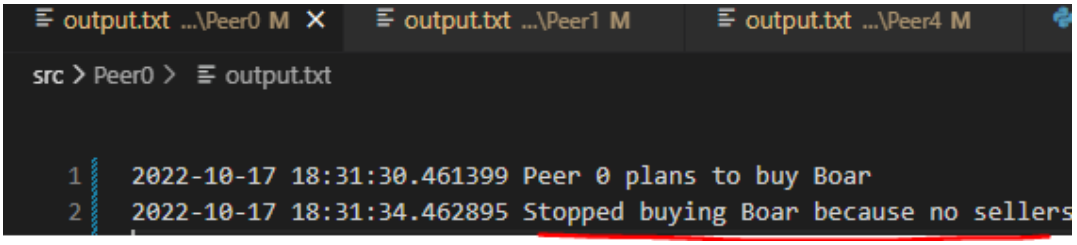
7. If good is not present



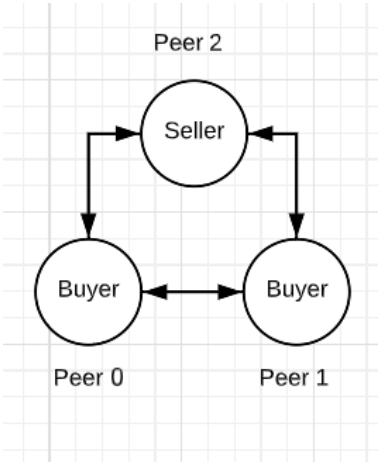
Peer 1 (Seller):

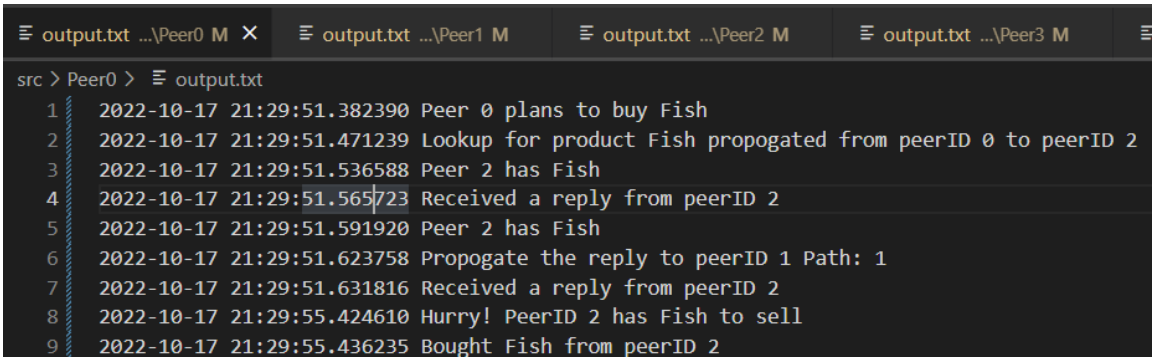


Peer 0 (Buyer):



8. If 2 buyers want the same good then requests get accepted one after the other because of the lock





output.txt ...\Peer0 M | output.txt ...\Peer1 M | output.txt ...\Peer2 M | output.txt ...\Peer3 M

src > Peer1 > output.txt

```
2022-10-17 21:29:51.387842 Peer 1 plans to buy Fish
2022-10-17 21:29:51.470541 Lookup for product Fish propogated from peerID 1 to peerID 2
2022-10-17 21:29:51.516130 Peer 2 has Fish
2022-10-17 21:29:51.539907 Received a reply from peerID 2
2022-10-17 21:29:51.569962 Peer 2 has Fish
2022-10-17 21:29:51.594643 Propogate the reply to peerID 0 Path: 0
2022-10-17 21:29:51.644610 Received a reply from peerID 2
2022-10-17 21:29:55.447271 Hurry! PeerID 2 has Fish to sell
2022-10-17 21:29:55.455627 Bought Fish from peerID 2
```

output.txt ...\Peer0 M | output.txt ...\Peer1 M | output.txt ...\Peer2 M

src > Peer2 > output.txt

```
2022-10-17 21:29:48.383344 Selling Fish: 4 Unit(s)
2022-10-17 21:29:55.417335 Remaining Fish: 3 Unit(s)
2022-10-17 21:29:55.436070 Remaining Fish: 2 Unit(s)
2022-10-17 21:30:02.488201 Remaining Fish: 1 Unit(s)
2022-10-17 21:30:04.464856 Remaining Fish: 0 Unit(s)
2022-10-17 21:30:04.474314 Selling new good Fish: 6 Unit(s)

```

9. If graph is fully connected

```
Distributed Systems/Lab/Lab-1/Cloned/CS677/src$ python3 test.py 9
Graph is not fully connected!
```

Performance

Response Time [sec/req]

The time to send a lookup request, find the appropriate seller and receive the reply back from the seller through the same path.

Average Response Time[sec/req]: The average of all response times of all potential sellers for a product sold by a buyer.

Insights: Relationship and Dependencies:
I) The response time and average response time depends on client wait time. If the client wait time is reduced, it is possible that no potential seller has been identified. If the client wait time is increased, many same sellers would be added to potential sellers which would end up increasing average response time. It is important to set the client wait time correctly, we have set it to 4 after various trial and error.
II) The average response time also depends on the hop count, if the hop count increases, more potential sellers can be identified and this can lead to increasing average response time.

Case of 1 potential seller:

```
2022-10-19 10:34:35.953285 Peer 0 plans to buy Salt
2022-10-19 10:34:35.960060 Peer 1 has Salt
2022-10-19 10:34:35.963414 Received a reply from peerID 1
2022-10-19 10:34:39.960214 Hurry! PeerID 1 has Salt to sell
2022-10-19 10:34:39.960884 Bought Salt from peerID 1
The response time of the seller chosen : 0.009721
The average response time for buying Salt: 0.009721
```

Case of 2 potential sellers:

```
2022-10-19 10:34:27.943230 Peer 3 plans to buy Salt
2022-10-19 10:34:27.957811 Peer 4 has Salt
2022-10-19 10:34:27.958818 Peer 2 has Salt
2022-10-19 10:34:27.961346 Received a reply from peerID 4
2022-10-19 10:34:27.962047 Received a reply from peerID 2
2022-10-19 10:34:31.954815 Hurry! PeerID 4 has Salt to sell
2022-10-19 10:34:31.955490 Bought Salt from peerID 4
The response time of the seller chosen : 0.017156
The average response time for buying Salt: 0.0175115
```

Latency:

The latency is measured based on the time when a client executes a RPC call till it returns. Below we have calculated the average latency for 1000 requests.

Case of 1 Seller:

```
1/Cloned/CS677/src$ python3 test.py 10
Running on: localhost
Testcase: 10
Neighbor Graph:
[False, True, False, False, False, False]
[True, False, True, False, False, False]
[False, True, False, True, False, False]
[False, False, True, False, True, False]
[False, False, False, True, False, True]
[False, False, False, False, True, False]
Hopcount:3
Marketplace is live! Check output.txt in PeerID directory to check the logging

Average latency of peer 2: 0.026163424999999983 (sec/req)
Average latency of peer 3: 0.027020772000000005 (sec/req)
Average latency of peer 1: 0.016471876999999996 (sec/req)
Average latency of peer 4: 0.022476200000000016 (sec/req)
Average latency of peer 2: 0.021439093499999978 (sec/req)
Average latency of peer 3: 0.023464429500000012 (sec/req)
```

Case of 2 Sellers:

```
ystems/Lab/Lab-1/Cloned/CS677/src$ python3 test.py 11
Running on: localhost
Testcase: 11
Neighbor Graph:
[False, True, False, False, False, False]
[True, False, True, False, False, False]
[False, True, False, True, False, False]
[False, False, True, False, True, False]
[False, False, False, True, False, True]
[False, False, False, False, True, False]
Hopcount:3
Marketplace is live! Check output.txt in PeerID directory to check the logging

Average latency of peer 1: 0.010397093999999992 (sec/req)
Average latency of peer 2: 0.011999331999999996 (sec/req)
Average latency of peer 3: 0.015992180999999984 (sec/req)
Average latency of peer 4: 0.013512766999999998 (sec/req)
Average latency of peer 1: 0.00955692849999999 (sec/req)
Average latency of peer 2: 0.011014073999999983 (sec/req)
Average latency of peer 3: 0.014992287500000001 (sec/req)
```

We can clearly see that as we increase the number of sellers, the latency reduces since the number of hops to find the seller reduces.

Separate Machine Deployment

We have added machines which specify the ip address of the servers being used. We have divided the peer servers equally among the machines. If there are an odd number of machines, the last machine will have less peer servers.

Initial Idea: We decided to use 2 machines connected on the same wifi, localhost of the one where we run the code and localhost of another machine connected on the same wifi. The one where we run the code is where the logging would take place. The peerServerList will be accordingly created as per the 2 machines, the same logic of the initial port number + peerId is applied here as well.

Challenges faced: As we tried to connect to localhost on another machine, we faced socket error and we couldn't connect to the host. The ping was successful but the program couldn't run.

```
prachitiparkar@prachiti src % python3 peer.py 4
Number of nodes: 4
Roles: Buyer Seller Seller Buyer
Neighbor Graph:
[False, True, False, False]
[True, False, True, True]
[False, True, False, True]
[True, True, True, False]
Hopcount:2
Marketplace is live! Check output.txt in PeerID directory to check the logging

Note: Peers are 0 indexed

Master machine is running on: localhost
Failed to connect to host. Please check host: http://192.168.1.42:49153
Failed to connect to host. Please check host: http://192.168.1.42:49153
```

To run:

1. Change deployOnLocalhost to False in src/peer.py
2. Change the machines ip address to the ip address you want to connect
3. Command - python3 peer.py #numberofpeers