CS77 - LAB 2 Asterix and the Trading Post

Team Members

Sanjana Radhakrishna - sanjanaradha@umass.edu

Akhil Reddy Anthireddy - aanthireddy@umass.edu

Design

In this project, we have implemented a fully connected network where each peer knows the address and port of all the other peers for communication. We have assigned each peer a role - buyer, seller and trader. Each seller will have their own item to sell and we have defined a price and amount for each item and the quantity of that item for each seller.

We have used the Pyro library for the communication between all remote objects and sending the RPC request calls.

Leader Election -

We have used 3 methods for the initial leader election, detection of leader failure and reelection -

initial_leader_election():

initial_leader_election() method will try to find the peer with the highest process id. We have used a variant of the bully algorithm to do this. Each peer will check if the peer with id greater than it is present or not. Example, Peer 1 checks if Peer 2 is present, then Peer 2 checks for Peer 3 and so on. If we have 6 nodes, then this process stops at Peer 6 and it is elected as leader and then its type is changed to "trader". And everyone is notified this peer is the trader. Note that initially we assume we have a reliable connection where all the nodes are in running state.

check_liveliness(trader_id):

Now that everyone knows who the trader is, all the other peers keep on checking at each buy/sell request if the leader(trader) is alive using the "**_PyroBind()**" to the trader using the trader's host and port. If any of the peers during these _PyroBind calls detects that there is no response from the leader, then all the other running processes are stopped and then **elect_leader(trader_id)** is called.

elect_leader(trader_id):
After we know the leader/ trader is down, we run the **elect_leader()** method and using the variant of the bully algorithm, each peer will send _PyroBind calls to all the peers which have process id greater than it. If the peer finds out there is at least one running peer above it. It stops the leader election call. When a peer does not receive any response during _PyroBind calls, it will elect itself as a leader and all the other peers are notified that this peer is the new leader

Clock Synchronization Implementation -

We have used the vector clocks for synchronization which is a python list(vector). And the vector clock algorithm is as follows -
update_vector_clock(peer_vec1, peer_vec2):
We have used the standard vector clock update algorithm, where each peer will have its own vector clock. When a request comes from Peer 1 to Peer 2, then we try to get the max of two vectors- peer_vec1 and peer_vec2 and update the Peer 2 vector clock value by 1.

All the peers will have their own vector clock which is initialized like - [0, 0, 0, 0, 0, 0]. When all the clock values are 0, then the trader will try to pick a buyer and seller by their id order.

After some time, we have different vector clock values for all peers. Now when multiple buy requests come in, all the requests are queued by the trader, Then the trader will identify the buyer with least clock value (which means it came first in order) and then pick the suitable seller for this buyer with least clock value and then complete the transaction. This way all queued requests are processed in a fixed order.

Transactions -

We have implemented methods for processing all the buyer/seller transactions -

register_products(self, seller_id);
register_products() method will register all seller info like the item, quantity and price with the trader. After multiple transactions, Whenever a seller is out of items to sell, he will again call the register_products() with the trader to register the new product and its quantity. The list of current sellers available is stored in sellers.txt file.

buy(self, buyer_id, product, buyer_clock):

The buyer calls the buy() method on the trader to request for a product. All the concurrent buy requests are queued with the help of priority queue on the vector clock. And one seller is also selected based on the sellers list we have created and his vector clock value. The trader then using the vector clocks identifies the first buyer and first seller and completes the transaction. Then subsequently we call update_buyer() and update_seller() to notify the buyer and seller that the transaction is complete.

update_buyer(self, trader_id):
This method will send acknowledgement to the buyer that the transaction is done and then the buyer will make another buy request with his updated vector clock .

update_seller(self, trader_id):
This method will send acknowledgement to the seller that the transaction is done and received then the seller will again wait in the list to sell the next item .

Transaction accounts maintenance -

We have created a python dictionary which maintains all the transactions information and looks like this -

```
Peer(id=0,
host="localhost",
neighbours=[1, 2, 3, 4, 5],
peer_type=PeerType.SELLER,
item=Item.SALT,
available_item_quantity=3,
amt_earned=0,
amt_spent=0,
commission=0,
vect_clock=[0, 0, 0, 0, 0, 0],
trader=-1,
port=get_free_port(),
)
```

All the peers info is stored in the above format in the dictionary and it is updated after each transaction.

We are also writing this dictionary to a **peers.json** file and updating it after every transaction so that in case of leader failure, we can retrieve all the data from this **peers.json** file
We are writing all the logged statements to the **result.txt** file.

Design Tradeoffs:

1) Updating transaction logs by the trader to a file is not done parallely. That is, the transactions and updating the logs is done sequentially. We would expect the CPU jobs and I/O jobs to be done parallely in an ideal scenario.

2) We have assumed our connections to be reliable and we are manually stopping the leader to trigger the election algorithm instead of automating it.

3) We have considered all the peers to be ran on the same machine. Hence this same application cannot be used when we run it on multiple machines

3) Every peer when it detects the leader is down is calling the elect_leader method so quickly, it looks like the leader is being updated by everyone parallely in real time.

Improvement/ Extensions:
1) When a new leader is being elected, we have stopped all the other processes. And also we put a lock on our elect_leader method. Hence, all the processes do not run leader elect requests parallely but sequentially. This can be improved

2) By running the application on multiple machines, we can increase the number of peers to a higher number and execute the transactions in a faster way.

3) We could run the update transaction in parallel while other transactions are in process. Our idea is to run these two actions in two different threads.

Running the Program :

1) How to run the Program ?

- Ensure sellers.txt file is empty
- From cli, run this command: **python3 main.py -N 6**

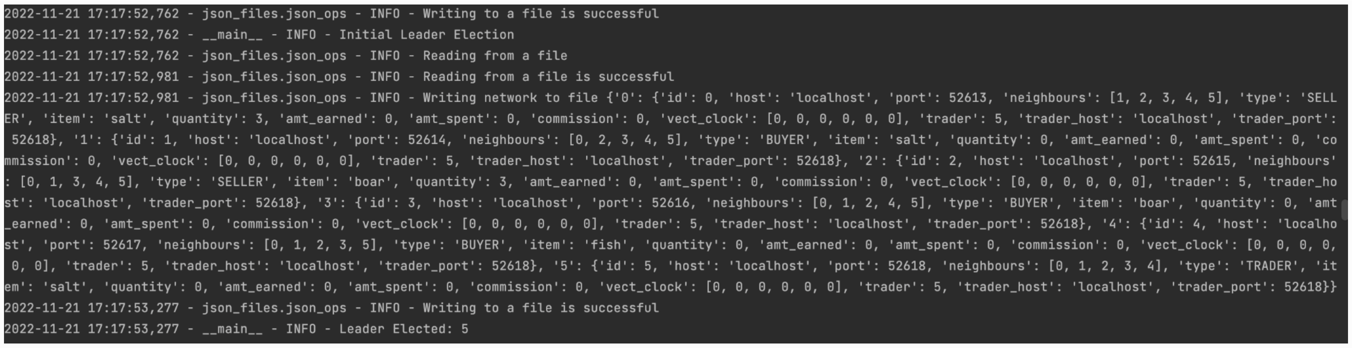
N - number of peers

2) Checking for leader election
- **Note down the trader_host and trader_port number from the logs**
- **Run “sudo lsof -i :<trader_port>” and note the process_id of the trader from the output**

- **Then, run “kill -9 <process_id>” to kill the trader and then check the logs or terminal to verify that the leader election algorithm has been triggered.**

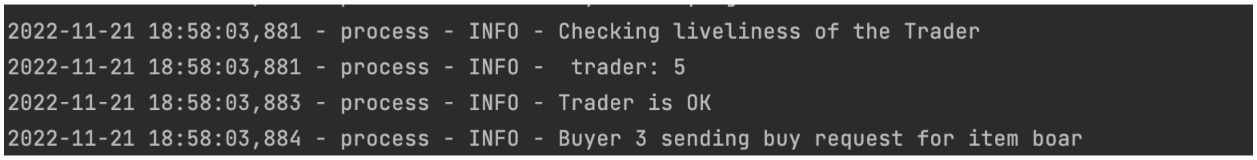
Test cases verified:

1) We have first verified that, immediately after all the peers are up and running, the leader election algorithm is run to elect the leader/trader among a bunch of buyers and sellers. From the below image we can verify the type of peer with highest id - Peer 5 has been changed to **“Trader”** after leader election



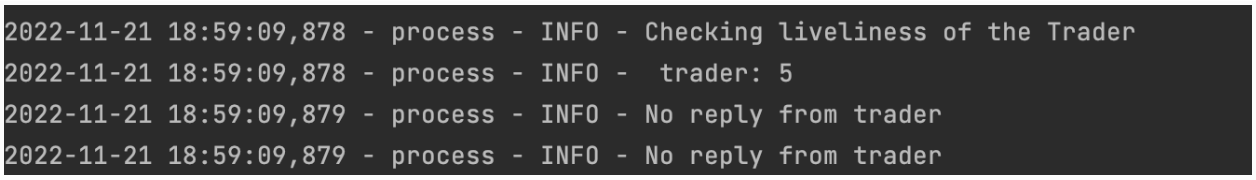
```
2022-11-21 17:17:52,762 - json_files.json_ops - INFO - Writing to a file is successful
2022-11-21 17:17:52,762 - __main__ - INFO - Initial Leader Election
2022-11-21 17:17:52,762 - json_files.json_ops - INFO - Reading from a file
2022-11-21 17:17:52,981 - json_files.json_ops - INFO - Reading from a file is successful
2022-11-21 17:17:52,981 - json_files.json_ops - INFO - Writing network to file {'0': {'id': 0, 'host': 'localhost', 'port': 52613, 'neighbours': [1, 2, 3, 4, 5], 'type': 'SELL
ER', 'item': 'salt', 'quantity': 3, 'amt_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port':
 52618}, '1': {'id': 1, 'host': 'localhost', 'port': 52614, 'neighbours': [0, 2, 3, 4, 5], 'type': 'BUYER', 'item': 'salt', 'quantity': 0, 'amt_earned': 0, 'amt_spent': 0, 'co
mmission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port': 52618}, '2': {'id': 2, 'host': 'localhost', 'port': 52615, 'neighbours'
: [0, 1, 3, 4, 5], 'type': 'SELLER', 'item': 'boar', 'quantity': 3, 'amt_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_ho
st': 'localhost', 'trader_port': 52618}, '3': {'id': 3, 'host': 'localhost', 'port': 52616, 'neighbours': [0, 1, 2, 4, 5], 'type': 'BUYER', 'item': 'boar', 'quantity': 0, 'amt
_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port': 52618}, '4': {'id': 4, 'host': 'localho
st', 'port': 52617, 'neighbours': [0, 1, 2, 3, 5], 'type': 'BUYER', 'item': 'fish', 'quantity': 0, 'amt_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0,
 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port': 52618}, '5': {'id': 5, 'host': 'localhost', 'port': 52618, 'neighbours': [0, 1, 2, 3, 4], 'type': 'TRADER', 'it
em': 'salt', 'quantity': 0, 'amt_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port': 52618}}
2022-11-21 17:17:53,277 - json_files.json_ops - INFO - Writing to a file is successful
2022-11-21 17:17:53,277 - __main__ - INFO - Leader Elected: 5
```

2) We are checking if the trader is alive at regular intervals

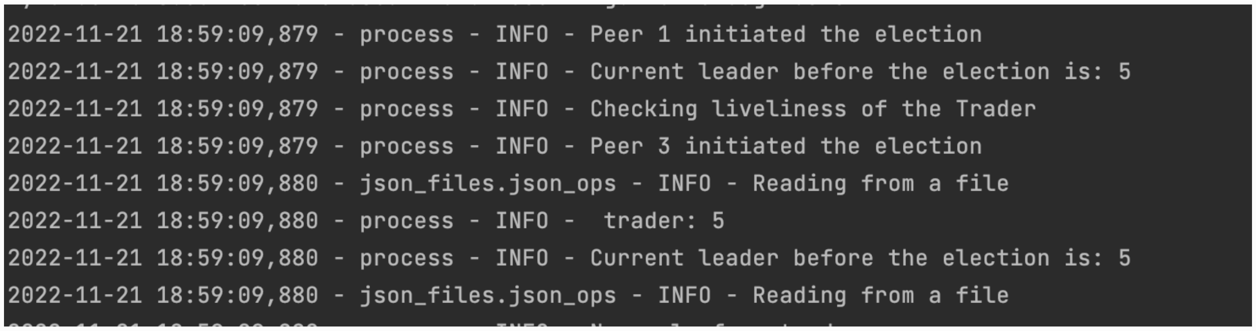


```
2022-11-21 18:58:03,881 - process - INFO - Checking liveliness of the Trader
2022-11-21 18:58:03,881 - process - INFO -  trader: 5
2022-11-21 18:58:03,883 - process - INFO - Trader is OK
2022-11-21 18:58:03,884 - process - INFO - Buyer 3 sending buy request for item boar
```

3) we have verified Leader election initiated by all peers when trader goes down For example, in the below images we can see Peer 1 and Peer 3 have started the election

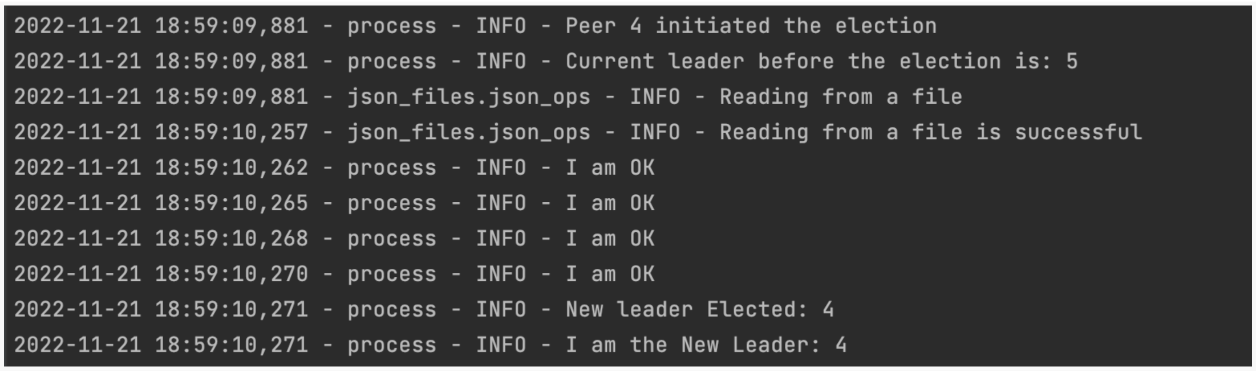


```
2022-11-21 18:59:09,878 - process - INFO - Checking liveliness of the Trader
2022-11-21 18:59:09,878 - process - INFO -  trader: 5
2022-11-21 18:59:09,879 - process - INFO - No reply from trader
2022-11-21 18:59:09,879 - process - INFO - No reply from trader
```



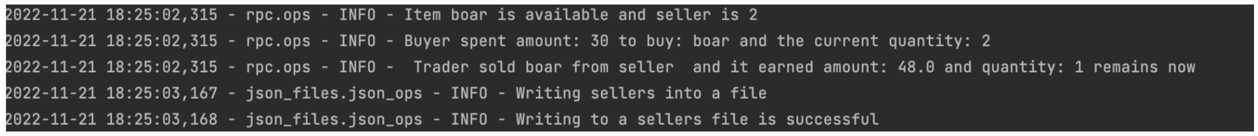
```
2022-11-21 18:59:09,879 - process - INFO - Peer 1 initiated the election
2022-11-21 18:59:09,879 - process - INFO - Current leader before the election is: 5
2022-11-21 18:59:09,879 - process - INFO - Checking liveliness of the Trader
2022-11-21 18:59:09,879 - process - INFO - Peer 3 initiated the election
2022-11-21 18:59:09,880 - json_files.json_ops - INFO - Reading from a file
2022-11-21 18:59:09,880 - process - INFO -  trader: 5
2022-11-21 18:59:09,880 - process - INFO - Current leader before the election is: 5
2022-11-21 18:59:09,880 - json_files.json_ops - INFO - Reading from a file
```

4) Check if the newly elected leader is able to inform all other peers about its win and pass its id to the other peers. Peer 4 did the election and realized it is the peer with the highest process id after Peer 5 is down. Hence it became the leader.



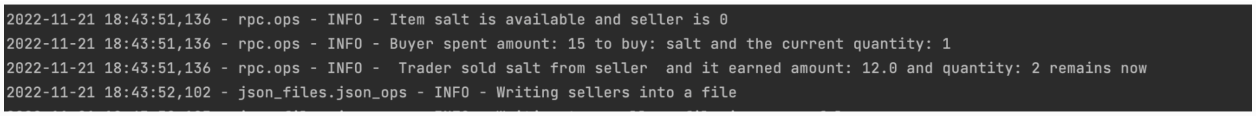
```
2022-11-21 18:59:09,881 - process - INFO - Peer 4 initiated the election
2022-11-21 18:59:09,881 - process - INFO - Current leader before the election is: 5
2022-11-21 18:59:09,881 - json_files.json_ops - INFO - Reading from a file
2022-11-21 18:59:10,257 - json_files.json_ops - INFO - Reading from a file is successful
2022-11-21 18:59:10,262 - process - INFO - I am OK
2022-11-21 18:59:10,265 - process - INFO - I am OK
2022-11-21 18:59:10,268 - process - INFO - I am OK
2022-11-21 18:59:10,270 - process - INFO - I am OK
2022-11-21 18:59:10,271 - process - INFO - New leader Elected: 4
2022-11-21 18:59:10,271 - process - INFO - I am the New Leader: 4
```

5) We have verified if the trader is able to sell the goods requested to the buyer.



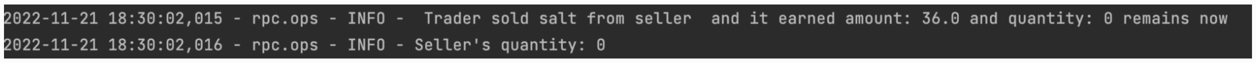
```
2022-11-21 18:25:02,315 - rpc.ops - INFO - Item boar is available and seller is 2
2022-11-21 18:25:02,315 - rpc.ops - INFO - Buyer spent amount: 30 to buy: boar and the current quantity: 2
2022-11-21 18:25:02,315 - rpc.ops - INFO -  Trader sold boar from seller  and it earned amount: 48.0 and quantity: 1 remains now
2022-11-21 18:25:03,167 - json_files.json_ops - INFO - Writing sellers into a file
2022-11-21 18:25:03,168 - json_files.json_ops - INFO - Writing to a sellers file is successful
```

6) We have also ensured that the trader notifies the corresponding seller about its item being sold. We can see from the below image that the amount earned by the seller and quantity remaining is also updated

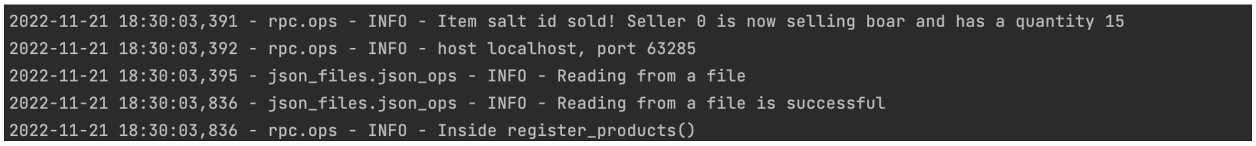


```
2022-11-21 18:43:51,136 - rpc.ops - INFO - Item salt is available and seller is 0
2022-11-21 18:43:51,136 - rpc.ops - INFO - Buyer spent amount: 15 to buy: salt and the current quantity: 1
2022-11-21 18:43:51,136 - rpc.ops - INFO -  Trader sold salt from seller  and it earned amount: 12.0 and quantity: 2 remains now
2022-11-21 18:43:52,102 - json_files.json_ops - INFO - Writing sellers into a file
```

7) Whenever a seller does not have any items to sell, then we have verified that the seller picks another item randomly and again registers with the trader.We can observe from below images that after the quantity remained became 0, seller 0 picked another item “boar” with a new quantity. Then it registers this details with the trader

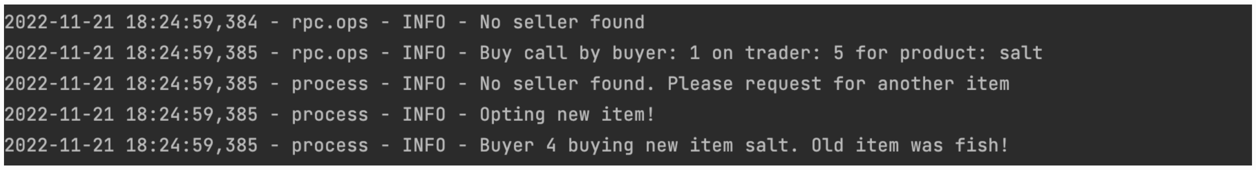


```
2022-11-21 18:30:02,015 - rpc.ops - INFO -  Trader sold salt from seller  and it earned amount: 36.0 and quantity: 0 remains now
2022-11-21 18:30:02,016 - rpc.ops - INFO - Seller's quantity: 0
```



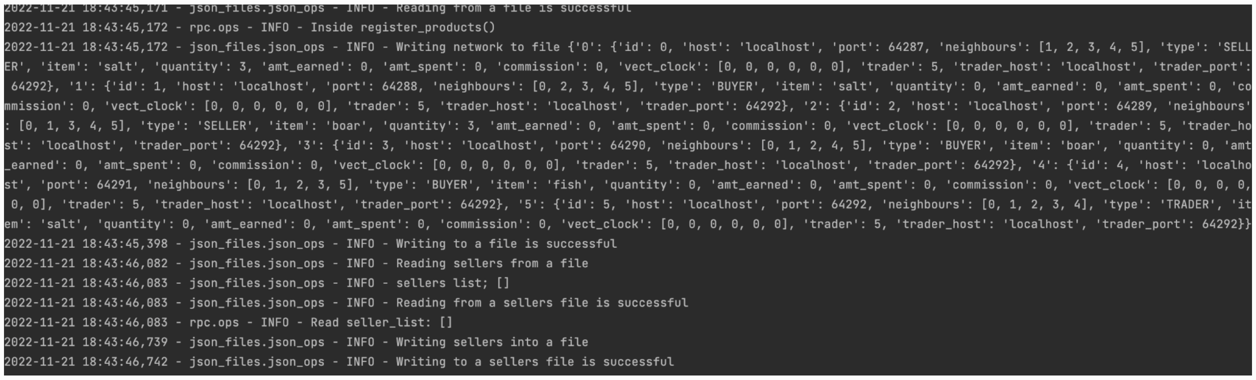
```
2022-11-21 18:30:03,391 - rpc.ops - INFO - Item salt id sold! Seller 0 is now selling boar and has a quantity 15
2022-11-21 18:30:03,392 - rpc.ops - INFO - host localhost, port 63285
2022-11-21 18:30:03,395 - json_files.json_ops - INFO - Reading from a file
2022-11-21 18:30:03,836 - json_files.json_ops - INFO - Reading from a file is successful
2022-11-21 18:30:03,836 - rpc.ops - INFO - Inside register_products()
```

8) We have checked that if so suitable seller is found for the buyer, buyer will change his item and then again raise buy request



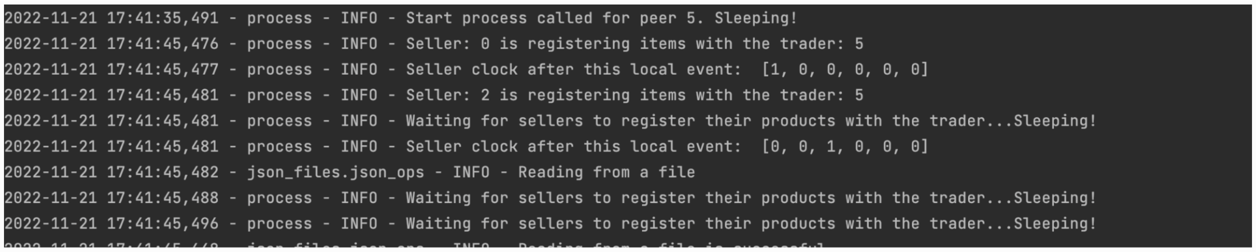
```
2022-11-21 18:24:59,384 - rpc.ops - INFO - No seller found
2022-11-21 18:24:59,385 - rpc.ops - INFO - Buy call by buyer: 1 on trader: 5 for product: salt
2022-11-21 18:24:59,385 - process - INFO - No seller found. Please request for another item
2022-11-21 18:24:59,385 - process - INFO - Opting new item!
2022-11-21 18:24:59,385 - process - INFO - Buyer 4 buying new item salt. Old item was fish!
```

9) Verifying that the transaction logs are getting updated properly and saved to a file.



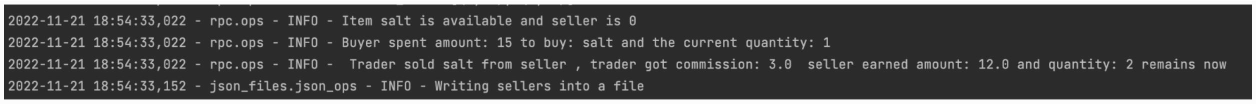
```
2022-11-21 18:43:45,171 - json_files.json_ops - INFO - Reading from a file is successful
2022-11-21 18:43:45,172 - rpc.ops - INFO - Inside register_products()
2022-11-21 18:43:45,172 - json_files.json_ops - INFO - Writing network to file {'0': {'id': 0, 'host': 'localhost', 'port': 64287, 'neighbours': [1, 2, 3, 4, 5], 'type': 'SELL
ER', 'item': 'salt', 'quantity': 3, 'amt_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port':
 64292}, '1': {'id': 1, 'host': 'localhost', 'port': 64288, 'neighbours': [0, 2, 3, 4, 5], 'type': 'BUYER', 'item': 'salt', 'quantity': 0, 'amt_earned': 0, 'amt_spent': 0, 'co
mmission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port': 64292}, '2': {'id': 2, 'host': 'localhost', 'port': 64289, 'neighbours'
: [0, 1, 3, 4, 5], 'type': 'SELLER', 'item': 'boar', 'quantity': 3, 'amt_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_ho
st': 'localhost', 'trader_port': 64292}, '3': {'id': 3, 'host': 'localhost', 'port': 64290, 'neighbours': [0, 1, 2, 4, 5], 'type': 'BUYER', 'item': 'boar', 'quantity': 0, 'amt
_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port': 64292}, '4': {'id': 4, 'host': 'localho
st', 'port': 64291, 'neighbours': [0, 1, 2, 3, 5], 'type': 'BUYER', 'item': 'fish', 'quantity': 0, 'amt_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0,
 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port': 64292}, '5': {'id': 5, 'host': 'localhost', 'port': 64292, 'neighbours': [0, 1, 2, 3, 4], 'type': 'TRADER', 'it
em': 'salt', 'quantity': 0, 'amt_earned': 0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0, 0, 0, 0], 'trader': 5, 'trader_host': 'localhost', 'trader_port': 64292}}
2022-11-21 18:43:45,398 - json_files.json_ops - INFO - Writing to a file is successful
2022-11-21 18:43:46,082 - json_files.json_ops - INFO - Reading sellers from a file
2022-11-21 18:43:46,083 - json_files.json_ops - INFO - sellers list; []
2022-11-21 18:43:46,083 - json_files.json_ops - INFO - Reading from a sellers file is successful
2022-11-21 18:43:46,083 - rpc.ops - INFO - Read seller_list: []
2022-11-21 18:43:46,739 - json_files.json_ops - INFO - Writing sellers into a file
2022-11-21 18:43:46,742 - json_files.json_ops - INFO - Writing to a sellers file is successful
```

10) Verifying that clock values of sellers are being updated before sending the request -



```
2022-11-21 17:41:35,491 - process - INFO - Start process called for peer 5. Sleeping!
2022-11-21 17:41:45,476 - process - INFO - Seller: 0 is registering items with the trader: 5
2022-11-21 17:41:45,477 - process - INFO - Seller clock after this local event:  [1, 0, 0, 0, 0, 0]
2022-11-21 17:41:45,481 - process - INFO - Seller: 2 is registering items with the trader: 5
2022-11-21 17:41:45,481 - process - INFO - Waiting for sellers to register their products with the trader...Sleeping!
2022-11-21 17:41:45,481 - process - INFO - Seller clock after this local event:  [0, 0, 1, 0, 0, 0]
2022-11-21 17:41:45,482 - json_files.json_ops - INFO - Reading from a file
2022-11-21 17:41:45,488 - process - INFO - Waiting for sellers to register their products with the trader...Sleeping!
2022-11-21 17:41:45,496 - process - INFO - Waiting for sellers to register their products with the trader...Sleeping!
2022-11-21 17:41:45,668 - json_files.json_ops - INFO - Reading from a file is successful
```

11) We have also checked that the trader is properly receiving his commission and then the transaction logs are being updated

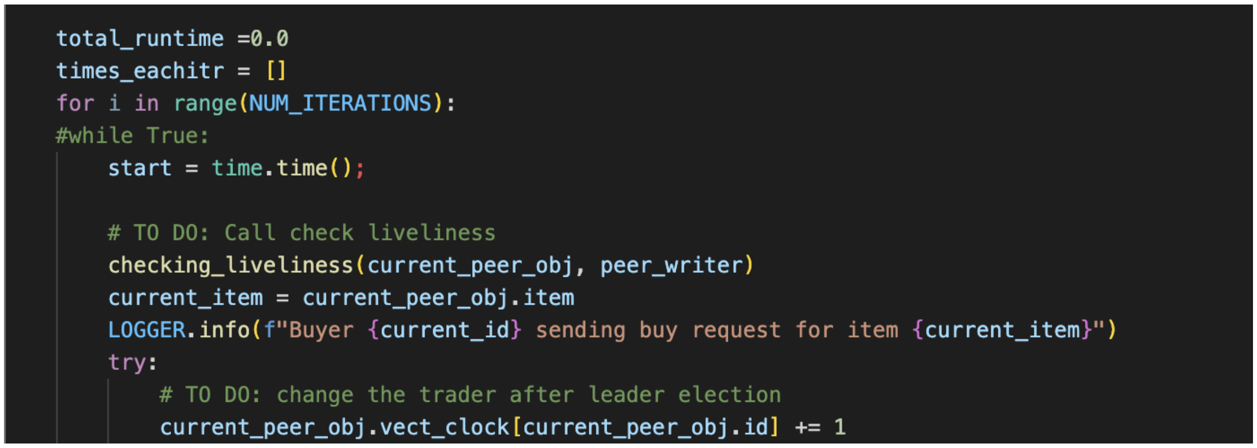


```
2022-11-21 18:54:33,022 - rpc.ops - INFO - Item salt is available and seller is 0
2022-11-21 18:54:33,022 - rpc.ops - INFO - Buyer spent amount: 15 to buy: salt and the current quantity: 1
2022-11-21 18:54:33,022 - rpc.ops - INFO -  Trader sold salt from seller , trader got commission: 3.0  seller earned amount: 12.0 and quantity: 2 remains now
2022-11-21 18:54:33,152 - json_files.json_ops - INFO - Writing sellers into a file
```

Performance Analysis and Results -

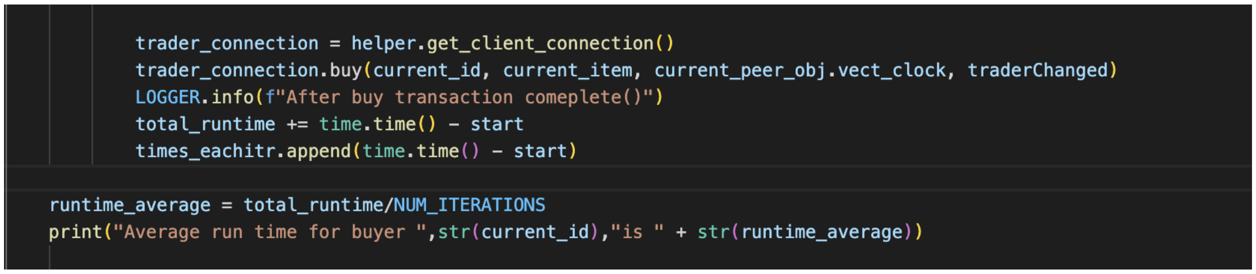
How we evaluated the response time -

We have used the time module of python to do this. We started the timer before the beginning of each lookup iteration and then measured the response time after the buyer gets a response from the seller. Then we average the response time for a fixed number of iterations. This response time method applies for both the evaluation metrics mentioned below.



```python
total_runtime =0.0
times_eachitr = []
for i in range(NUM_ITERATIONS):
#while True:
    start = time.time();

    # TO DO: Call check liveliness
    checking_liveliness(current_peer_obj, peer_writer)
    current_item = current_peer_obj.item
    LOGGER.info(f"Buyer {current_id} sending buy request for item {current_item}")
    try:
        # TO DO: change the trader after leader election
        current_peer_obj.vect_clock[current_peer_obj.id] += 1
```

We start the timer before initiating a buy request and then measure timer after the response for each iteration after transaction complete and then take average for each buyer which will look like -



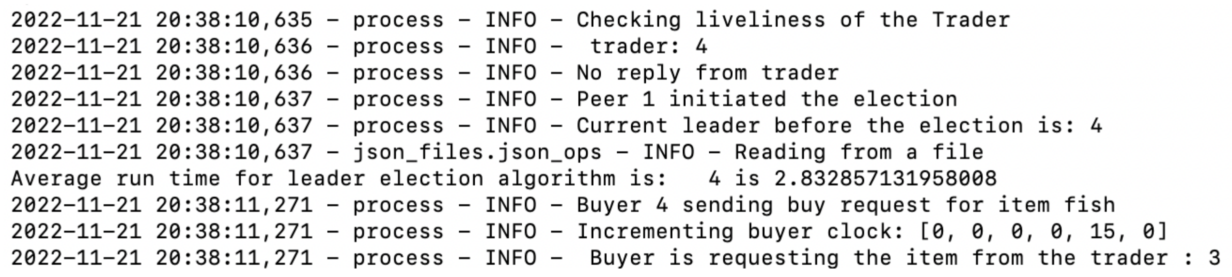
```python
            trader_connection = helper.get_client_connection()
            trader_connection.buy(current_id, current_item, current_peer_obj.vect_clock, traderChanged)
            LOGGER.info(f"After buy transaction comeplete()")
            total_runtime += time.time() - start
            times_eachitr.append(time.time() - start)

    runtime_average = total_runtime/NUM_ITERATIONS
    print("Average run time for buyer ",str(current_id),"is " + str(runtime_average))
```

However in our actual code, we have not run for fixed iterations but the program will always keep running until some external interrupt comes in.

Analysis -

1) Average time for the leader election -

The average run time for the leader election algorithm is around **2.83 seconds**. This is the scenario when we have 6 peers. And as discussed from other results below, we will see that average response time for 1 buyer is around **0.7 seconds**. So the time taken for the leader election is almost 4 times the response time for buyers. This leader election will slow down the system drastically.

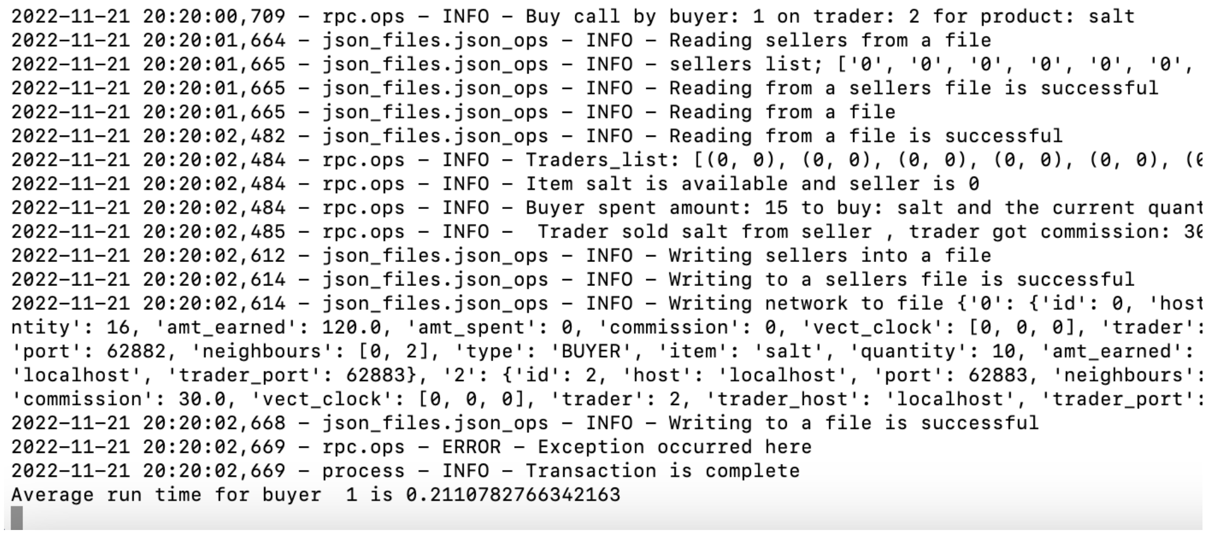


```
2022-11-21 20:38:10,635 - process - INFO - Checking liveliness of the Trader
2022-11-21 20:38:10,636 - process - INFO -  trader: 4
2022-11-21 20:38:10,636 - process - INFO - No reply from trader
2022-11-21 20:38:10,637 - process - INFO - Peer 1 initiated the election
2022-11-21 20:38:10,637 - process - INFO - Current leader before the election is: 4
2022-11-21 20:38:10,637 - json_files.json_ops - INFO - Reading from a file
Average run time for leader election algorithm is:   4 is 2.832857131958008
2022-11-21 20:38:11,271 - process - INFO - Buyer 4 sending buy request for item fish
2022-11-21 20:38:11,271 - process - INFO - Incrementing buyer clock: [0, 0, 0, 0, 15, 0]
2022-11-21 20:38:11,271 - process - INFO -  Buyer is requesting the item from the trader : 3
```

2) Response time per buyer when numbers of buyers is increasing -

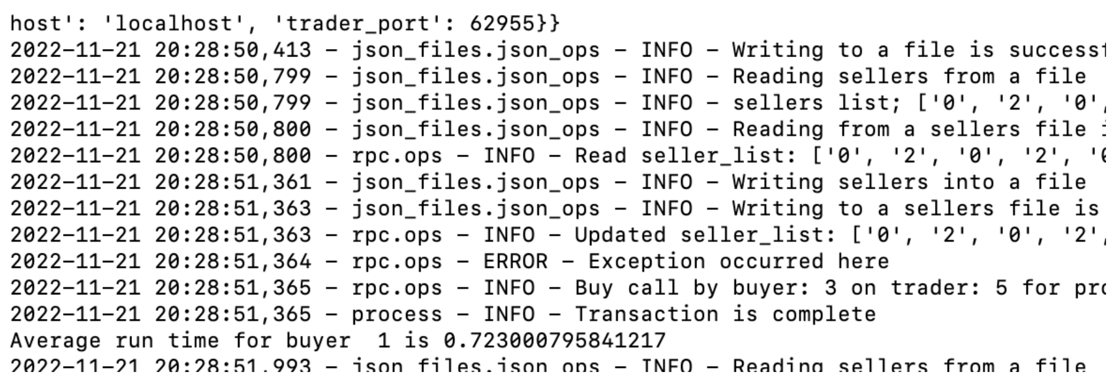
Here we have considered a setup where there is only 1 seller and we are considering 3 scenarios - 2 buyers, 3 buyers, and 4 buyers. For each scenario, we have run 25 iterations and for each iteration, we calculated the response time for buyers and took an average for those 25 iterations.

The average response time for buyer 1 is when there is only one buyer, one seller and one trader is around **0.211 sec** as shown below -



```
2022-11-21 20:20:00,709 - rpc.ops - INFO - Buy call by buyer: 1 on trader: 2 for product: salt
2022-11-21 20:20:01,664 - json_files.json_ops - INFO - Reading sellers from a file
2022-11-21 20:20:01,665 - json_files.json_ops - INFO - sellers list; ['0', '0', '0', '0', '0', '0',
2022-11-21 20:20:01,665 - json_files.json_ops - INFO - Reading from a sellers file is successful
2022-11-21 20:20:01,665 - json_files.json_ops - INFO - Reading from a file
2022-11-21 20:20:02,482 - json_files.json_ops - INFO - Reading from a file is successful
2022-11-21 20:20:02,484 - rpc.ops - INFO - Traders_list: [(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (
2022-11-21 20:20:02,484 - rpc.ops - INFO - Item salt is available and seller is 0
2022-11-21 20:20:02,484 - rpc.ops - INFO - Buyer spent amount: 15 to buy: salt and the current quant
2022-11-21 20:20:02,485 - rpc.ops - INFO -  Trader sold salt from seller , trader got commission: 3
2022-11-21 20:20:02,612 - json_files.json_ops - INFO - Writing sellers into a file
2022-11-21 20:20:02,614 - json_files.json_ops - INFO - Writing to a sellers file is successful
2022-11-21 20:20:02,614 - json_files.json_ops - INFO - Writing network to file {'0': {'id': 0, 'host
ntity': 16, 'amt_earned': 120.0, 'amt_spent': 0, 'commission': 0, 'vect_clock': [0, 0, 0], 'trader':
'port': 62882, 'neighbours': [0, 2], 'type': 'BUYER', 'item': 'salt', 'quantity': 10, 'amt_earned':
'localhost', 'trader_port': 62883}, '2': {'id': 2, 'host': 'localhost', 'port': 62883, 'neighbours':
'commission': 30.0, 'vect_clock': [0, 0, 0], 'trader': 2, 'trader_host': 'localhost', 'trader_port':
2022-11-21 20:20:02,668 - json_files.json_ops - INFO - Writing to a file is successful
2022-11-21 20:20:02,669 - rpc.ops - ERROR - Exception occurred here
2022-11-21 20:20:02,669 - process - INFO - Transaction is complete
Average run time for buyer  1 is 0.2110782766342163
```

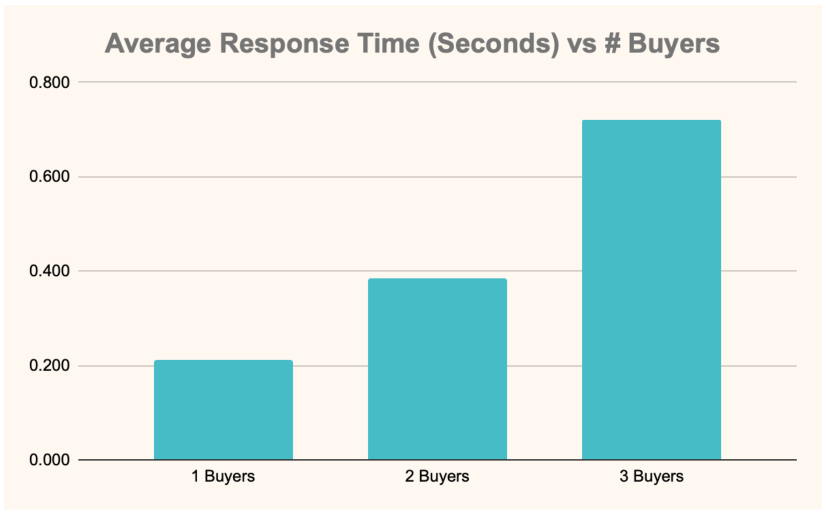
Similarly we have increased the number of buyers and sellers and then verified the time taken to process. For example, for the image given below number of buyers is 3, number of sellers is 2 and average response time for buyer 1 has increased to **0.722 sec**. As the number of buyers is increased then the response time for each buyer is also increased.



```
host': 'localhost', 'trader_port': 62955}}
2022-11-21 20:28:50,413 - json_files.json_ops - INFO - Writing to a file is successf
2022-11-21 20:28:50,799 - json_files.json_ops - INFO - Reading sellers from a file
2022-11-21 20:28:50,799 - json_files.json_ops - INFO - sellers list; ['0', '2', '0',
2022-11-21 20:28:50,800 - json_files.json_ops - INFO - Reading from a sellers file i
2022-11-21 20:28:50,800 - rpc.ops - INFO - Read seller_list: ['0', '2', '0', '2', '(
2022-11-21 20:28:51,361 - json_files.json_ops - INFO - Writing sellers into a file
2022-11-21 20:28:51,363 - json_files.json_ops - INFO - Writing to a sellers file is
2022-11-21 20:28:51,363 - rpc.ops - INFO - Updated seller_list: ['0', '2', '0', '2',
2022-11-21 20:28:51,364 - rpc.ops - ERROR - Exception occurred here
2022-11-21 20:28:51,365 - rpc.ops - INFO - Buy call by buyer: 3 on trader: 5 for pro
2022-11-21 20:28:51,365 - process - INFO - Transaction is complete
Average run time for buyer  1 is 0.723000795841217
2022-11-21 20:28:51,993 - json_files.json_ops - INFO - Reading sellers from a file
```

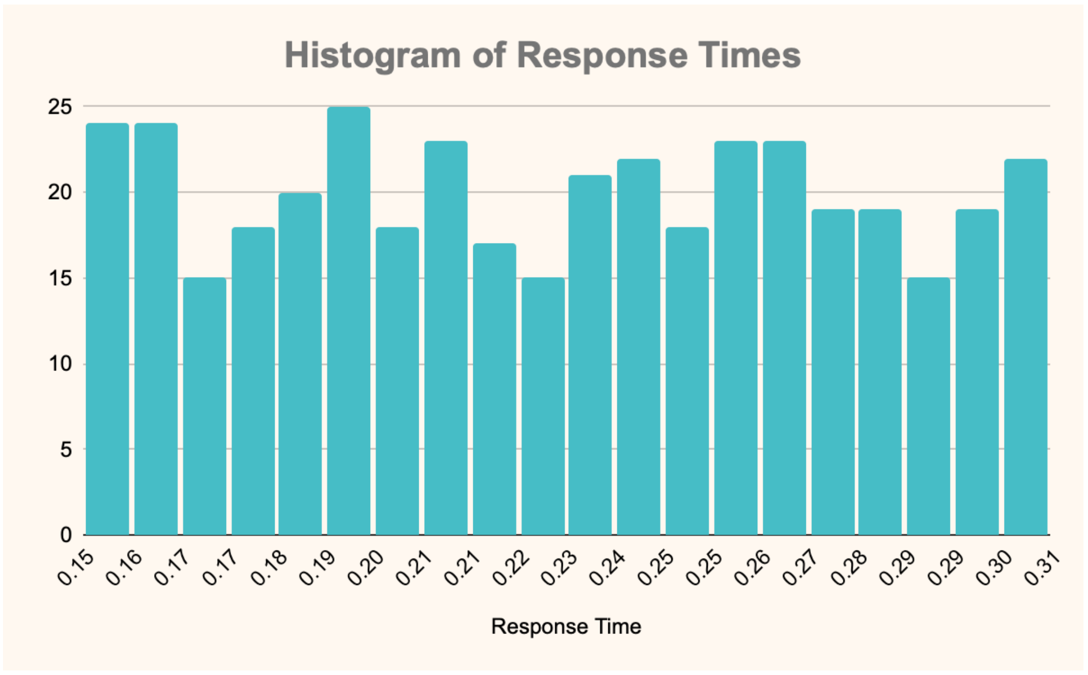
Results for other scenarios have been published below -

	Average response time in seconds		
	1 Buyers	**2 Buyers**	**3 Buyers**
	0.211	0.36	0.692
		0.41	0.751
			0.722
Average	0.211	0.385	0.722



3) Average Response per client search request -

We have considered a scenario where there is only 1 buyer and 1 seller and 1 trader. Then we have used python's time.time module to calculate the response time after each request. We have stored the response time for 450 requests and created a histogram which shows the distribution for those 450 requests as below in sec. The response time per client request was around 0.2 to 0.3 seconds.



We have observed that in case we kill a leader, then the elect leader algorithm is triggered which will slow down the process by a bit but if we consider time taken for 450 requests, then one leader failure won't affect the runtime much but more number of leader failures can become a bottleneck.