COMPSCI 677: Distributed and Operating Systems
Lab 2

Jenish Bajracharya, Aditya Chaloo

November 21, 2024

1 Introduction

This report documents Asterix and the Trading Post, a trading system that simulates the exchange of three goods: salt, boar, and fish, between a trader, sellers, and buyers. Initially, each seller deposits their goods with the leader to ensure that the leader's inventory is not empty when a buy request is made. A buyer initiates a buy request for one of the goods by sending a request to the trader. The trader processes the buy request by comparing vector clocks. Once a purchase is confirmed, the trader updates its inventory, sends a buy confirmation message to the buyer, and also notifies the seller. The seller then updates their local inventory. Buyers, sellers, and traders interact via UDP sockets. Additionally, the system incorporates fault tolerance to handle scenarios where a trader goes offline. A bully algorithm is implemented to elect a new leader with the highest ID, ensuring that the state of the defaulted leader is inherited by the new leader. Once the election is complete, the defaulted leader is demoted to either a buyer or a seller, and the buy/sell process resumes under the new leader.

2 System Architecture

2.1 Network Design

Our network is fully connected, meaning each peer is connected to every peer in the network. This allows for easy broadcasting of messages during election and helps reduce latency by avoiding peer discovery. Each peer is identified by its `peer_id` and their network address (`'localhost', 5000 +` {`peer_id`}).

```
Peer 0 (leader) connected to peers [1, 2, 3, 4, 5]
Peer 1 (buyer) connected to peers [0, 2, 3, 4, 5]
Peer 2 (buyer) connected to peers [0, 1, 3, 4, 5]
Peer 3 (seller) connected to peers [0, 1, 2, 4, 5]
Peer 4 (buyer) connected to peers [0, 1, 2, 3, 5]
Peer 5 (seller) connected to peers [0, 1, 2, 3, 4]
Peer 0 (leader) with item None listening on port 5000...
Peer 1 (buyer) with item None listening on port 5001...
Peer 2 (buyer) with item None listening on port 5002...
Peer 3 (seller) with item boar listening on port 5003...
Peer 4 (buyer) with item None listening on port 5004...
Peer 5 (seller) with item boar listening on port 5005...
```

Figure 1: Network Design with 6 peers, each peer connected to every other peer. The roles are assigned randomly however during setup `Peer0` is assigned as the leader.

2.2 Message Handling

The system employs a message-passing mechanism using sockets to facilitate interactions between each peer. Messages are of two classes:

1. Trading Messages: These types of messages are responsible for managing the buy/sell processes within the system.

2. Election Messages: These classes of message are responsible for handling the election process.

2.2.1 Trading Messages

- `UpdateInventoryMessage`$_{\texttt{Seller}\rightarrow\texttt{Leader}}$: This message is sent from a seller to the leader to update the inventory. It includes details about the seller's ID, product information, stock availability, price, and a vector clock for consistency.

- `BuyMessage`$_{\texttt{Buyer}\rightarrow\texttt{Leader}}$: A buyer sends this message to the leader when they want to purchase a specific product. It specifies the buyer's ID and address, the product they wish to buy, the desired quantity, amount, and a unique request ID for tracking. A vector clock is also included which will involve in comparison of timestamps between the buyers as well as the trader.

- `SellConfirmationMessage`$_{\texttt{Leader}\rightarrow\texttt{Seller}}$: The leader sends this message to the seller to confirm the sale of a product to a buyer. It mirrors the details in the buy confirmation message but is tailored for the seller, confirming the transaction's success. It also includes the amount after deducting commission.

- `BuyConfirmationMessage`$_{\texttt{Leader}\rightarrow\texttt{Buy}}$: Sent by the leader to the buyer, this message confirms whether a purchase request was successful. It includes the original request ID, the status of the purchase (success or failure), the quantity approved for purchase, and updated vector clock data.

2.2.2 Election Messages

- `ElectionMessage`$_{\texttt{Peer}_i\rightarrow\texttt{Peer}_{j:j>i}}$: This message is sent by a peer to other peers with higher IDs during the leader election process. The peer who sends this message is randomly chosen when the leader fails. It signals the initiation of an election, indicating that the sender considers the current leader unresponsive or unavailable. Peers receiving this message participate in the election based on the bully algorithm logic.

- `OKMessage`$_{\texttt{Peer}_j\rightarrow\texttt{Peer}_i}$:Sent in response to an `ElectionMessage`, this message informs the sender that a peer with a higher ID is active and participating in the election process. It serves as a confirmation that the election process will continue and prevents multiple leaders from being elected simultaneously. The indices j and i refer to the message going in the reverse direction from $\texttt{peer}_j$ to $\texttt{peer}_i$, following the `ElectionMessage` received by $\texttt{peer}_j$.

- `LeaderMessage`$_{\texttt{Leader}\rightarrow\texttt{Peers}}$: Once a new leader is elected, this message is broadcasted by the newly chosen leader to inform all peers of their status. It ensures synchronization within the network and allows peers to update their local state to reflect the new leader.

2.3 Peer

The `Peer` class serves as the core component for each participant in the network. Each peer operates either as a buyer or a seller or a trader, with distinct functionalities tailored to their roles. We summarize the functionality for each peer bases on their roles via the following finite state machine. Additionally, for parallelizability each peer is euqipped with a thread pool parameterized by `MAX_WORKERS`.

2.3.1 Seller

Once the network is initialized and the roles have been assigned, the seller randomly picks an item to sell along with the quantity and price. Additionally it also starts it's local timer. The seller then sends the `UpdateInventoryMessage` to the leader. The seller then transitions into WaitForMessage state which is constantly monitoring for messages from the leader. If a sell confirmation message is received then it runs the `handle_sell_confirmation` method and updates its local inventory and reduces the stock by the quantity sold. Once the stock reaches zero, the seller randomly chooses another item to sell and transitions to

SendUpdateInventoryMessageToTrader state. When the seller is either in WaitForMessage or RunHandleSellConfirmation state and the timeout gets triggered, it randomly samples a random number from (0,1) and compares its value with the `UPDATEINVENTORYPROBABILITY`. If the value is less than the probability, the seller then samples another item to sell and sends the update inventory message to the leader. We use timer and `UPDATE_INVENTORY_PROBABILITY` to model the behavior of a seller as a seller can send items to buy at anytime not necessarily when the stock reaches zero. For all the messages that gets sent by the seller, their vector clock gets incremented and attached with the message. Additionally, we manage receiving and sending of messages via a thread pool for parallelizability.

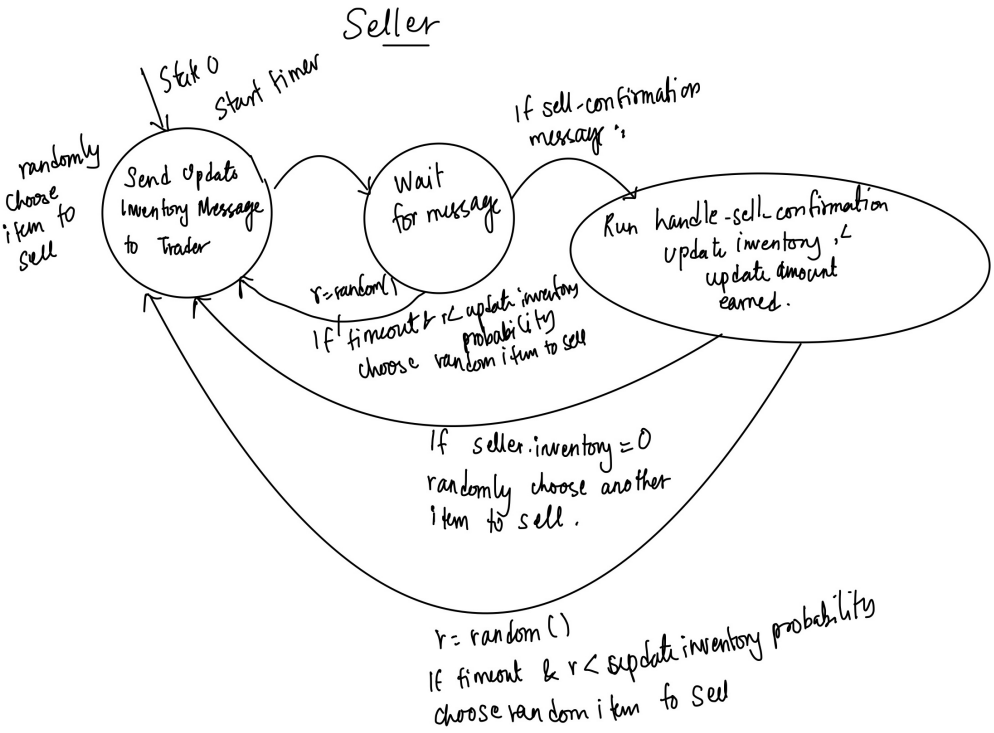


Figure 2: FSM for Seller

2.3.2 Buyer

Once the network is initialized and the roles have been assigned, the buyer waits for the initial inventory to be set with the leader. Once the setup is confirmed the buyer then chooses a random item and a random quantity to buy from the leader. It creates a buy messages and sends it to the leader and transitions to WATIFORCONFIRMATIONMESSAGE state. If the buy fails then the buyer proceeds to look for another item. If the buy succeed then the buyer can go ahead and buy another item with a probability `BUY_PROBABILITY`. Similar to the seller, the buyer increments and appends its vector clock with the messages.

2.3.3 Leader

The leader is responsible for executing all the buy and sell operation in the network so serves as a crucial component in our system. Receiving and sending of messages are handled via a thread pool but for simplicity we explain the working of the leader assuming a process with few threads. The leader initially waits for the sellers to set up their inventory with the leader via the `Update_Inventory_Message`. Once the setup is done it starts the election timer and then waits to receive other messages. If a `Update_Inventory_Message` is received, it transitions back to the initial state and updates its inventory. If a `Buy_Message` is received from a buyer, it runs the `handle_buy` method and sends buy confirmation and sell confirmation to the the buyer and the

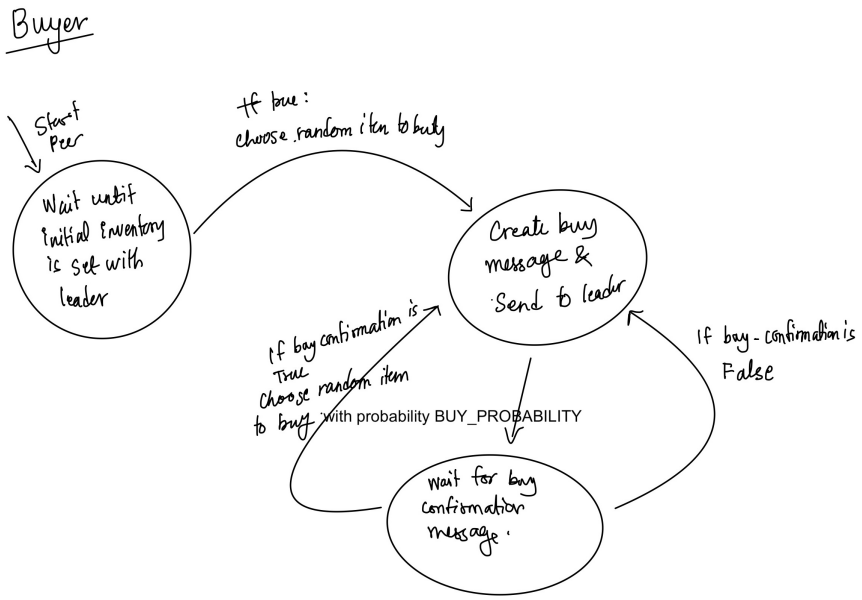


Figure 3: FSM for Buyer

seller. When the leader is in any of these states, there is another thread running that keeps track of the election timer, if the timer is triggered the the leader will fail with a probability `LEADER_FAILURE_PROBABILITY`. Once the leader fails, all operations are halted and its state is written to a pickle file which includes its inventory and the requests that need to be processed. These requests are stored the `Pending_Requests` array. Once the election is done the defaulted leader gets its role reassigned. The defaulted leader either gets demoted to either a buyer or a seller and transitions to the buyer or seller FSM.

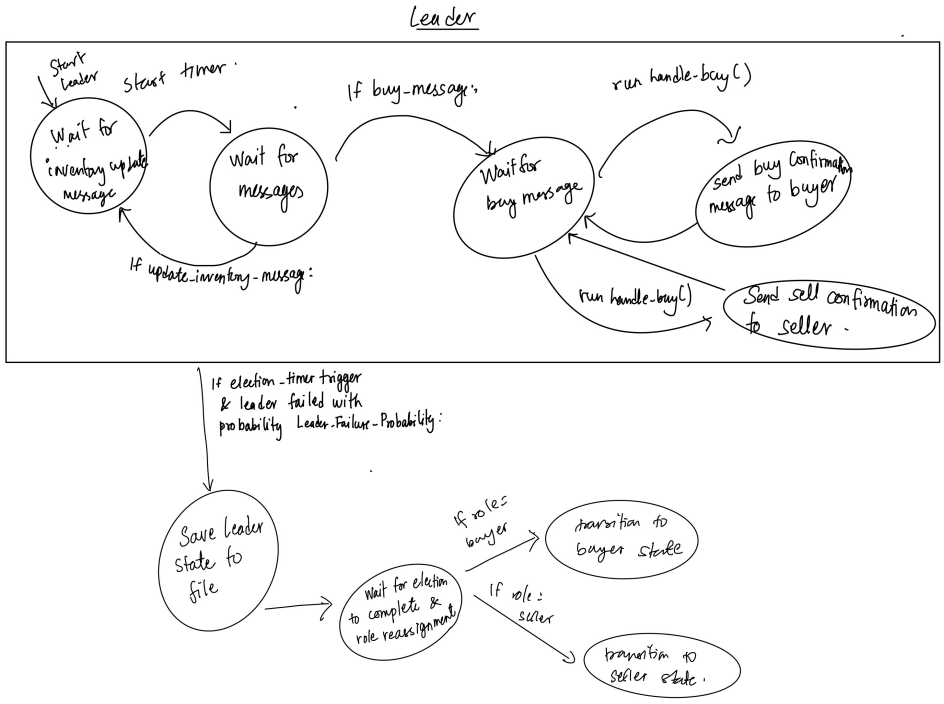


Figure 4: FSM for Leader

2.4 Election Monitor Thread

Each peer in our system is initialized and managed by `main.py`. The file contains a `monitor_election` function that detects simulates the failure of a leader when the leader's election timer gets triggered. Once the election timer gets triggered and the leader fails with `LEADER_FAILURE_PROBABILITY` the election monitor chooses a peer at random and initiates a election. If the leader crashes then operation in all peers are halted. This is achieved via locks. If a peer is chosen to initiate election, it initiates election bases on bully algorithm. Once a leader is declared, it send the `leader` message to every peer in the network. And each peer resumes its operations. The `main.py` also handles the reassigning of the defaulted leader to either a buyer or seller. The new leader reads the state from the pervious leader and repopulates the inventory and pending request array.

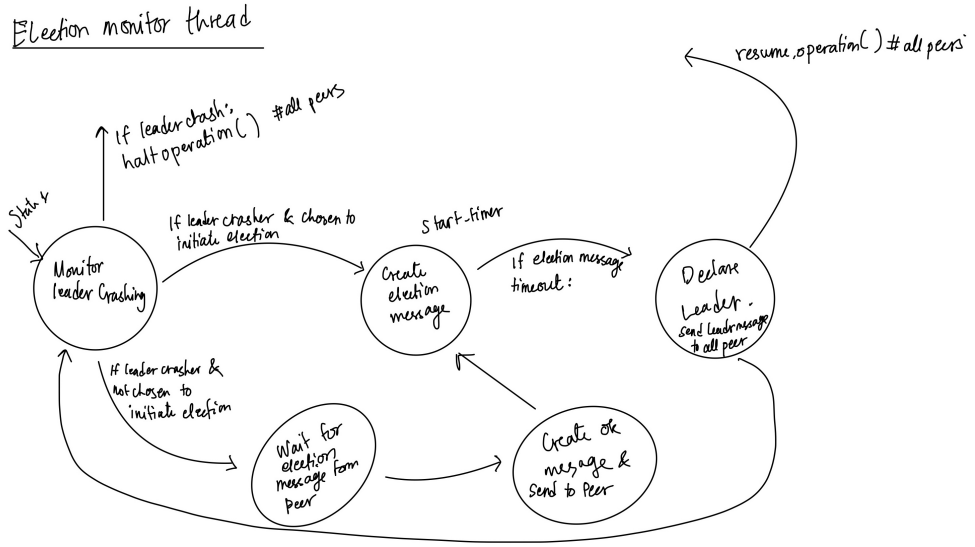


Figure 5: FSM for Election Monitor Sub Module

3 Testing

3.1 Running the Scripts

Run the `main.py` script with the desired number of peers as an argument. For example, to start a network with 6 peers:

```
python main.py 6
```

3.2 Test Case Categories

- **Functional Tests**: Verify that the system's functionalities work as intended.
- **Edge Case Tests**: Evaluate the system's behavior under unusual or extreme conditions.
- **Error Handling Tests**: Ensure that the system gracefully handles errors and exceptions.

NOTE: Terminal outputs are formatted as `[{peer_id} <output>]` which shows, the output is generated by peer with `peer_id`. Additionally we have not written test cases for buy-sell confirmation functionality because these are trivially visible from in the logs from any of the test cases below. Furthermore, one test case can have multiple test cases.

3.3 Functional Tests

1. **TC_01: Commission Going to Trader**

   - **Description**: Ensure that the trader (leader) receives a commission from sales.

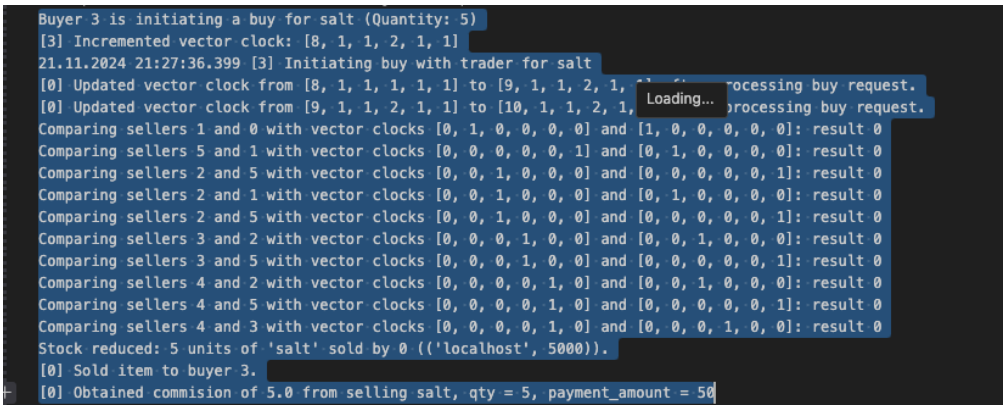


Figure 6: TC_01: Commission Going to Trader

   - **Expected Outcome**: The trader's total earnings are updated with commissions from sales.
   - **Result**: Buyer 3 initiates a buy for fish which is fulfilled by the leader and the leader receives a 10% commission (1) for a payment amount of 10.

2. **TC_02: Seller Gets His Money**

   - **Description**: Verify that sellers receive their share of the transaction after the trader deducts the commission.

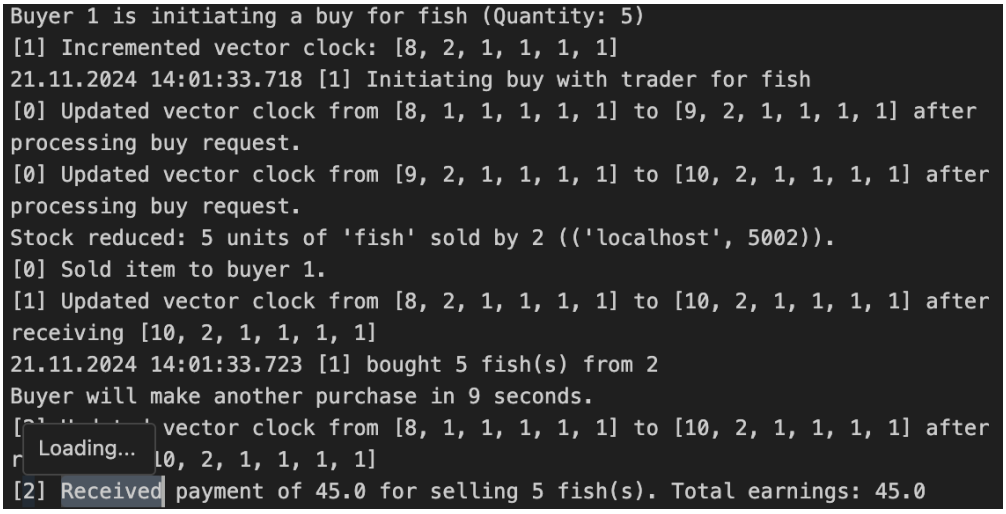


Figure 7: TC_02: Seller gets his money

   - **Expected Outcome**: The sellers receive their payment after successful sales, minus the trader's commission.
   - **Result**: Seller 2's accounts is credited with their share of the sell after deducting the commission.

3. **TC_03: Concurrent Request Handling**

   - **Description**: Ensure the system correctly handles multiple concurrent purchase requests.

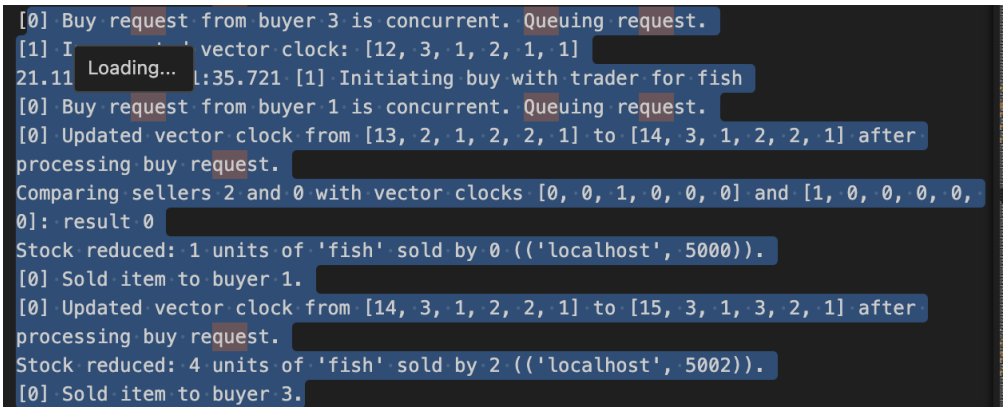


Figure 8: TC_03: Concurrent Request Handling

- **Expected Outcome**: The leader processes concurrent requests appropriately, queuing them if necessary, and handles them in order.
- **Result**: The leader identifies and handles concurrent requests, maintaining transaction order. The request from peer 3 is concurrent which gets added to the queue and then subsequently executed.

3.4 Edge Case Tests

1. **TC_04: No Available Sellers**

   - **Description**: Ensure the system handles scenarios where no sellers have the requested item or insufficient stock.

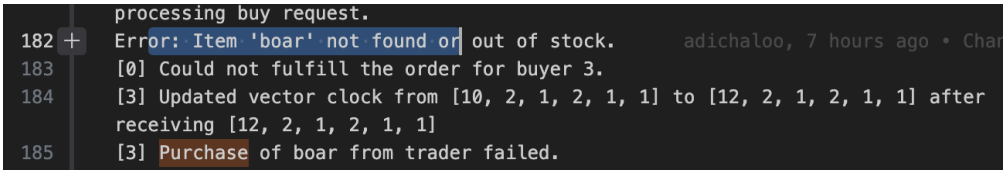


Figure 9: TC_04: No Available Sellers

   - **Expected Outcome**: The leader informs buyers that the purchase cannot be fulfilled due to unavailability.
   - **Result**: Buyers 3 attempting to purchase unavailable items receive error messages.

2. **TC_05: Election**

   - **Description**: Test the system's ability to elect a new leader when the current leader fails or steps down.
   - **Expected Outcome**: Correctness of bully algorithm, inventory saved to disk and new leader loads the inventory from disk. New leader announced to every peer. All buy and sell operation are halted when leader fails.
   - **Result**: Leader 0 failed, peer 2 initiates the election, after completion of election peer 5 becomes the leader. This shows the correctness of the bully algorithm. Additionally, inventory loaded from disk messages is displayed which shows that the state of the previous leader is inherited by the new leader. New loader is announced to every peer and every peer updates their leader attribute. Since no buy/sell messages are displayed, the operation are halted.

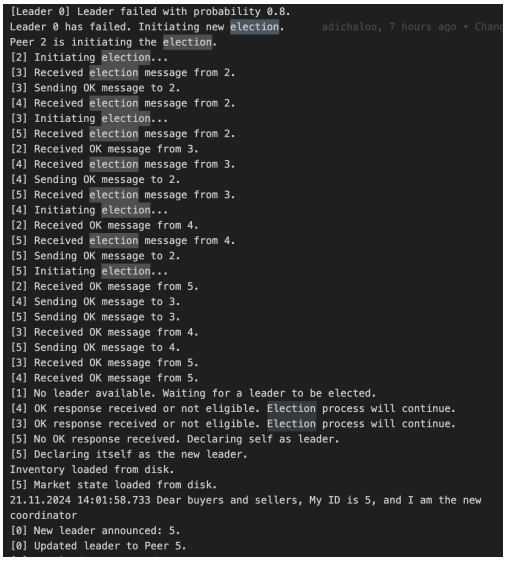
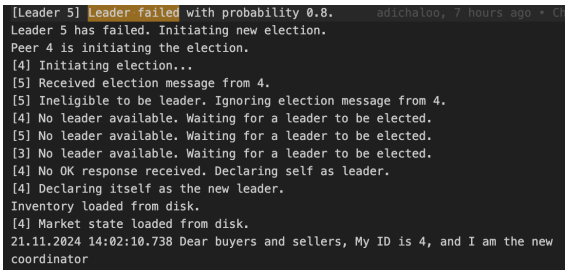


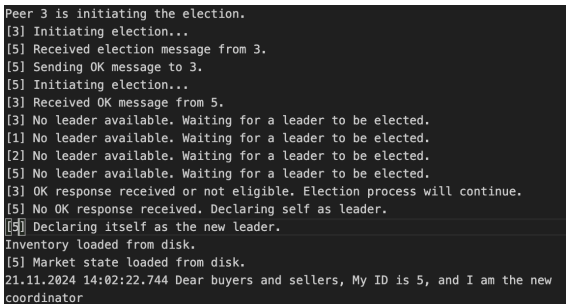
Figure 10: TC_05: Electing New Leader

3. **TC_06: Election Switching Between Only Highest Peer ID**

- **Description**: Assess the election process when only peers with the highest IDs are candidates, ensuring fair and consistent leader selection.



(a) 5 fails, 4 is elected as new leader



(b) 4 fails, 5 is elected as new leader

Figure 11: TC_06: Election Switching Between Only Highest Peer ID

- **Expected Outcome**: The leader role toggles consistently between peers with the highest IDs.
- **Result**: The trader/leader role alternates between PeerID 4 and PeerID 5. This is because, once leader 5 defaults, it gets demoted to a peer with the same id so during the next election, leader 5 gets elected hence alternates between 5 and 4.

4. **TC_07: The Previous Leader Also Participates in the Marketplace**

- **Description**: Examine the system's behavior when a former leader becomes a regular participant (seller or buyer) in the marketplace.
- **Expected Outcome**: Once the leader loses its position, it can continue as a regular participant in the marketplace.
- **Result**: The previous leader, such as PeerID 0, transitions to act as a seller and sends inventory message to leader which show that the previous leader has transitioned to being a seller in the marketplace.

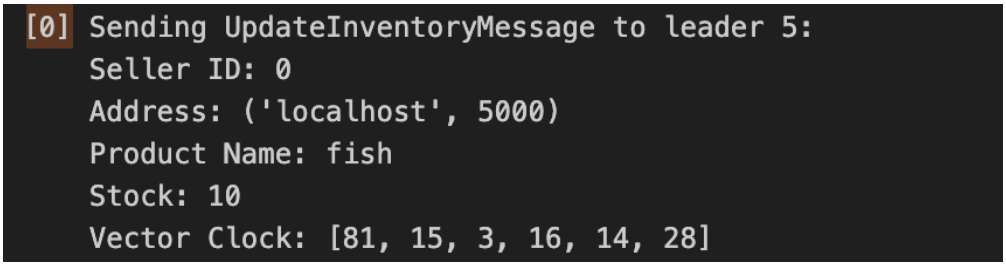


Figure 12: TC_07: The Previous Leader Also Participates in the Marketplace

5. **TC_08: Even After Trader Goes Down, Marketplace is Still Saved**
   - **Description**: Ensure that the marketplace state persists and operations continue smoothly even if the trader (leader) node fails.
   - **Expected Outcome**: After every successful transaction, the marketplace state is saved. When a new leader is elected, it loads the saved inventory and resumes trading.
   - **Result**: In figure 10 The new leader is elected peer 5, loads the saved inventory, and continues trading with buyers.
6. **TC_09: Concurrent Request Handling**
   - **Description**: Ensure that conflicts arising from simultaneous requests are managed correctly, and errors are handled without data corruption.

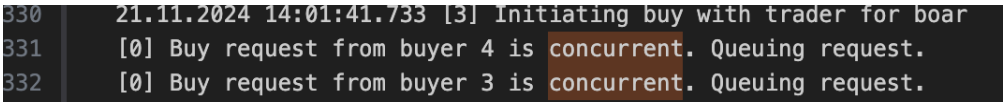


Figure 13: TC_03: Concurrent Request Handling

   - **Expected Outcome**: Requests that cannot be completed due to conflicts are queued. Once their timestamps are clear of conflicts, the trader executes the transactions.
   - **Result**: Conflicting requests are queued and later executed, ensuring no data corruption or transaction loss.

4 Evaluation

(E1) The concurrency tests were conducted to evaluate the average round trip time of all the buyers to get a response when there is election and there is no election. The tests were run with three different configurations of `MAX_TRANSACTIONS`: 10, 20, and 100 and election probability (`LEADER_FAILURE_PROBABILITY`). When leader fails with probability 0 election probability is 1 and vice versa.

(E2) We measure the response time for each request for various buyers in the system. In this scenario the election happens with probability one.

4.1 Test Configuration Constants

The following constants are used to configure the behavior of buyers and sellers within the P2P network:

```
# config.py
BUY_PROBABILITY = 0   # Probability that a buyer will continue buying after a successful
    purchase
SELLER_STOCK = 5      # Each seller starts with 5 items
MAX_TRANSACTIONS = 10/ 20/ 30/ 40  # NUMBER OF TRANSACTIONS A BUYER CAN DO BEFORE IT
    SHUTSDOWN
TIMEOUT = 0.1  #S
PRICE = 1
COMMISSION = 0.1

LEADER_FAILURE_PROBABILITY = 0 / 1  # 0 = No election, 1 = Election every 10 seconds.
TIME_QUANTUM = 10  # Time in seconds        # Timeout duration in seconds

UPDATE_INVENTORY_PROBABILITY = 0.3
```

Listing 1: Test Configuration Constants

4.2 Test Outputs

The following sections present the results of the concurrency tests conducted with different values of `MAX_TRANSACTIONS` with varied election probabilities.

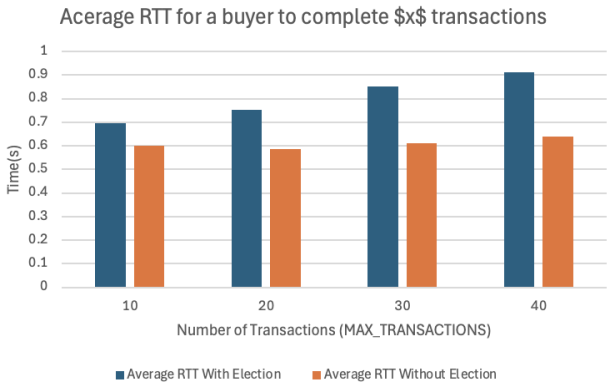
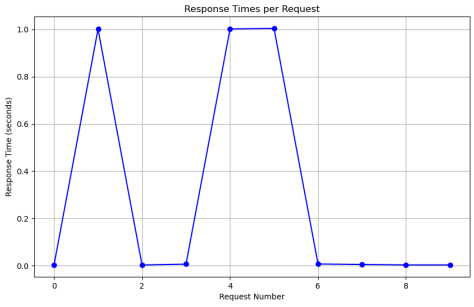
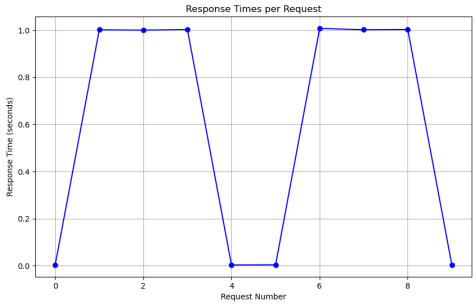


Figure 14: (E1) Average RTT fro 1 request with or without election.



(a) Buyer 1 Response time



(b) Buyer 2 Response time

Figure 15: (E2) Resopnse time for a particular request when the system is simulated with election.

4.3 Observations

(E1)As the election happens we observe that the time it takes to complete transaction increases. This behavior is expected as when election happens, all operation are halted, additionally, the newly elected leader has to load the state from disk which consumes time.

(E2) Where there is no election the response time is almost 0 for each request for each buyer however when there is election, a request takes time to get serviced. For example, the peeks represents the time it took for a request to get serviced when election happens.

5 Limitations, Design Trade-off, and Improvements

In this section, we describe the limitation(L) and design trade offs (T) we considered for our system. Additionally we present a few improvements (I) for the takeoffs and limitations.

1. (T1) UDP for Communication: We use UDP as the transport layer protocol for efficiency. UPD is suitable for running peers locally, however for large systems limitations of UDP over TCP might bottleneck the entire system.
2. (T2) Upon initialization of the network, the first peer with `peer_id` 0 is assigned as the leader.
3. Fully Connected Network
   - (T3) Our network is fully connected with each peer connected to every peer. We make this design choice to avoid peer discovery and easier broadcasting of messages.
   - (I1) A potential improvement could be have each peer connected to the leader and random other peers which depicts real-world scenario.
4. Peer Role
   - (T4) As per the assignment description, "a peer can be both a buyer or a seller, or only one of the two" our system assigns a Peer as either a buyer or a seller.
   - (I2) A potential improvement could be to have the peer be both the buyer and a seller. This scenario is indicative of a real world marketplace where a buyer is a seller and a seller is a buyer as well. However in this setting we would need to have additional logic to make sure that the seller of an object is not the buyer or the same object.
5. Switching of Leader
   - (T4) Since our system demotes the defaulted leader to the buyer or seller our leader keeps switching between peer 4 and peer 5.
   - (I2) A potential improvement could be to have the defaulted leader be online after some time modeled by a probability so that other peers also have the chance of becoming the leader.
6. Static Network
   - (L1) Our system does not support new peers joining the network once initialized.
   - (I3) A possible enhancement would be to allow dynamic peer addition, where new peers can join an already initialized network. This feature would improve system scalability and make it more representative of real-world networks.
7. Fault Tolerance
   - (L2) Our system is able to handle leader failing by initiating a election. If a buyer or a seller fails or goes offline unexpectedly, our system lacks fault tolerance mechanisms to handle this scenario. Pending requests are lost, and the system relies on timeouts and retries for recovery.

- (I4) To improve fault tolerance, the system could implement a distributed logging mechanism or checkpointing. This would also require the deployment of the Atomicity property, ensuring that all operations are completed fully or not at all. This approach would enhance reliability, ensuring consistency and continuity even in the event of unexpected peer failures or system crashes.