Lab 2 – Asterix and the Trading Post

Submission by:

Bishwas Dhakal - bdhakal@umass.edu

Ash Pai – agpai@umass.edu

Contents

1. How to run the program

Running Test case 1	`python3 main.py test_case1` This will create a network size of 7, out of which 5 of them are sellers and two buyers. This demonstrates an election algorithm, leader falling sick occasionally, reelection to elect new leader and two buyers buying two different products. A seller is capable of restocking a random product from the choice to if it runs out of stock.
Running Test Case 2	`python3 main.py test_case2` This will create a network size of 5, out of which 4 of them are sellers and one buyer. This demonstrates an election algorithm, a leader falling sick occasionally, reelection to elect a new leader and one buyer sending 1000 requests to buy a product. Peer 3 gets elected as first leader, as the Peer 4 is assigned sick when setting up the network.
Running Test Case 3	`python3 main.py test_case3` This will create a network size of 7, out of which 5 of them are sellers and two buyers. This demonstrates an election algorithm, leader falling sick occasionally, reelection to elect new leader and two buyers buying two same products. Concurrent buy requests are resolved using multicast Lamport's clock.
Normal Run, to simulate everything	`python3 main.py normal`
Normal Run, to simulate everything with N given peers	`python3 main.py normal <N>` E.g. `python3 main.py normal 10` This will create a 10 peers network and assign randomly buyers and sellers. Selects a random peer and starts an election using Bully Algorithm to elect a leader. This demonstrates an election, the leader falling sick occasionally, reelection to elect a new leader and buyers buying their respective products. Concurrent buy requests are resolved using multicast Lamport's clock. A seller is capable of restocking a random product from the choice to if it runs out of stock.

2. Introduction

This program simulates a peer to peer network that supports leader election and lamport's logical clock for concurrent requests. Each peer in the network is initialized either as a Buyer or a Seller. A randomly selected peer from the network initiates an election, election is carried out using Bully algorithm. The elected leader takes the role of the trader and refrains from its previous role as Buyer or Seller. The buyer and sellers will perform their respective roles as normal, and buyers and sellers will have their transaction mediated via the leader of the network. If the leader reports as sick, one of the active peers initiates a new election and elects a new leader via the bully algorithm. Any concurrent transactions that are received by the tradesmen will be resolved via the Lamport's clock timestamp.

3. Core Functionalities

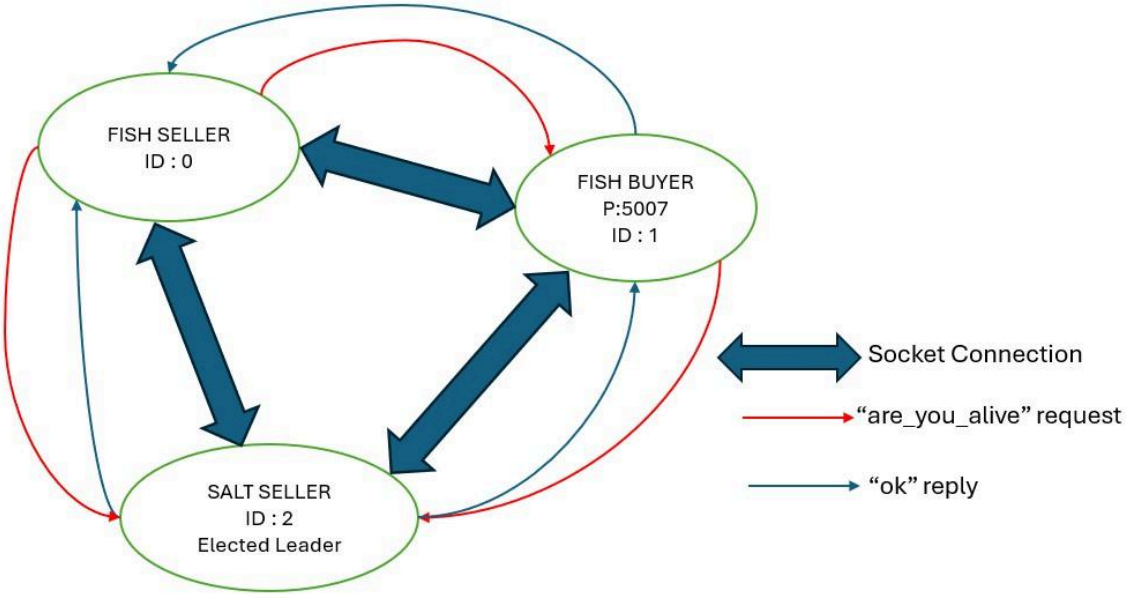


Figure 1: Example diagram to show P2P network demonstrating election with Bully Algorithm

Each of the peers is running on a separate process, and each of the processes is listening to its port as well as spinning a thread to handle any incoming request messages. One peer is communicating with another peer using socket connection. Each of the peers has its own logical clock that is used for handling concurrent buy requests.

3.1. Peer Class

The core of the program is the Peer class, representing a node in the P2P marketplace. Each peer has the following attributes:

- **peer_id**: A unique identifier for each peer.

- **role**: Defines whether the peer is a buyer or a seller.
- **product**: If the peer is a seller, this attribute specifies the product they have for sale.
- **neighbors**: A list of other peer objects to which the peer is connected.
- **stock**: The quantity of the product available for sellers.
- **alive**: Boolean describing whether the node is in service or not
- **cash_recieved:** if the node is a seller this int describes the amount of cash acquired from selling products. Each item price is set to 1$.
- **network_size:** number of other nodes in the network
- **request_already_sent:** used to determine if the node has responded to the election
- **leader:** used to determine if this node is the leader of the network
- **leader_id:** used to determine the ID of the leader node in the network
- **lamport_clock:** the timestamp used by each node to determine the order in which transactions should be performed by the leader.
- **request_queue**: the queue (sorted by lamport's timestamp) to know the order in which the transactions should be performed by the leader.

The class also maintains a global dictionary peers_by_id to map peer IDs to their corresponding objects, enabling peers to communicate by referencing their IDs.

3.2 Communication using sockets

Peers communicate with each other using TCP sockets:

- **listen_for_requests()**: Each peer runs a server socket to listen for incoming requests from neighbors.
- **handle_request()**: Once a connection is accepted, the peer processes the request, which can be to check if the peer is alive, run an election, buy product request, set the new leader by writing that into a shared file, request and provide sellers list of products for sale.
- **send_request()**: This method sends requests to the peer's neighbors. The request can either be to check if the peer is alive, run an election, buy product requests, set the new leader, request and provide sellers list of products for sale.

3.3. Message Handling

The peer system supports several types of messages(request):

3.3.1 Election

- **set_leader**: After the bully election algorithm completes the leader must be set and multicasted to all other members of the network. Since each peer is running in its own process when this message is received it will use the ‘leader.csv’ file to learn which node has been elected the leader
- **are_you_alive:** On network initialization or re-election the node issuing a election must send a message to peers with a higher peer_id and validate that they are alive. The receiver peer sends an ‘ok’ message to the sender if it is still active.
- **ok**: The peer starting the election will receive this message if a higher peer_id in the network has responded. If the peer receives this message it can safely assume that it is not the leader.

3.3.2 Market

- **buy**: A request sent by buyers to the leader indicating that they are interested in purchasing a product from the market. The leader will be responsible for checking the inventory (disk file) to see if the market is open and there is stock available to sell from a seller.
- **give_seller_list:** When a leader is elected on initialization it needs to get inventory from all the sellers.
- **selling_list:** Each seller must respond back to the leader with its peer_id, product_name, and stock.
- **Item_bought:** When an item is purchased from a seller the seller inventory and disk file needs to be updated. Each item is priced 1$ and when the leader sells an item to a buyer, the leader sends this request type to the seller to compensate for its item.
- **request_queue:** When the leader is handling multiple buy requests it must place all the buy requests into the queue and sort it by lamport's timestamp to know the order in which the buys should be handled.
- **restock_item:** When the seller runs out of its stock, it can randomly choose a product to sell and restock its count to 100.

2.3.2 Lamport Clock

- **multicast**: When a buy request is sent to the leader/tradesman in order for all the other peers in the network to update their own clock the request must be multicast across the network.

4. Areas of improvement

One of the areas of improvement is gracefully finishing all the threads spun by the processes. This can lead to connection errors while rerunning the program quickly after terminating it. This is because the handle request threads are not terminated gracefully. It can be resolved by

keeping a list of all the spun threads and at the end of the program calling join() on each of the threads to finish their execution.

A second area of improvement would include a more intelligent way for the leader to fall_sick(). In the current implementation the fall sick function is checked on a per-transaction basis. Meaning that for 1000 requests if we set a 5% fall sick rate there will be 50 (5% of 1000) elections taking place. This makes it challenging to test small network configurations as eventually there is only 1 leader remaining with no other active nodes. Additionally, it would have been helpful to implement reinstating a node after it becomes sick. This way nodes could rotate in and out of service. In the current model once a node is removed from service it stays removed from service.

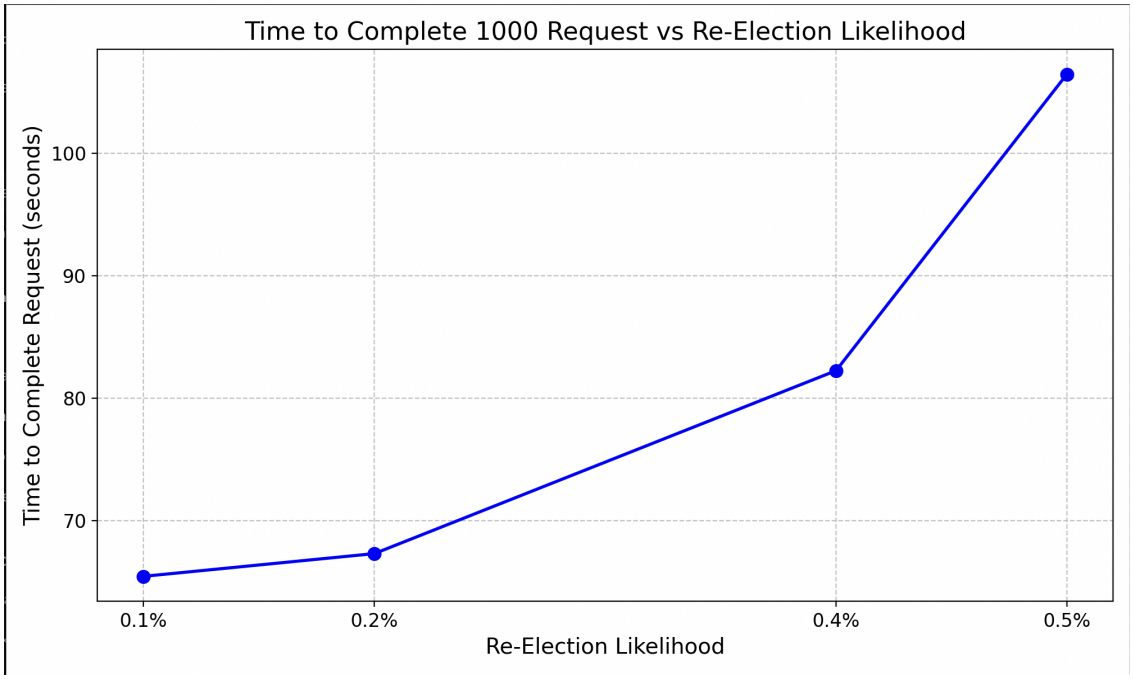
5. Design Trade-offs

In our implementation of the project the leader can call in sick at any point, handled with a probability that the leader could call itself sick, when the leader calls sick, it sets a flag election_in_progress and writes that into a shared file 'leader.csv'. Based on this flag, buyer peers can suspend their buy request until the election_in_progress gets cleared. The flag gets cleared when the election is complete. All the buy requests that have been sent before the flag is cleared continue to be processed by the current leader who is calling in sick until the request_queue is empty. Another approach might be to have the leader call in sick and hand off (could save into a shared file) all pending buy requests to the newly elected leader and this newly elected leader complete these buy requests. This could lead to a more graceful handling of the requests after the crashed leader node allowing for a seamless purchasing process for the buyer and seller nodes handled by the newly elected leader.

As the main process is unaware of the current leader, and each buyers buy request is sent to the current leader, everytime a new leader is elected, the new leader id is written into the shared file ‘leader.csv’, and main process is reading this file to figure out the current leader in the network.

6. Performance

To simulate 1000 sequential requests with varying likelihoods of reelection, an experiment was conducted with 5 nodes: 1 seller, 3 buyers, and a leader. In each transaction, there was a certain probability of the leader "falling sick" (i.e., becoming unavailable), triggering a reelection. The likelihood of reelection was kept small to ensure that enough nodes remained active in the network after a reelection, thus maintaining the stability of the system while still testing for leadership resilience.



Time in Seconds to Complete 1000 requests	Percentage Chance of Reelection
65.443	0.1%
67.300	0.2%
82.249	0.4%
106.468	0.5%

As anticipated, increasing the likelihood of reelection resulted in longer processing times for the 1000 requests. This outcome aligns with the design of the program, where all buy requests are paused during an election. A higher reelection likelihood meant more frequent elections within the 1000 requests, which in turn increased the time needed to complete the requests.

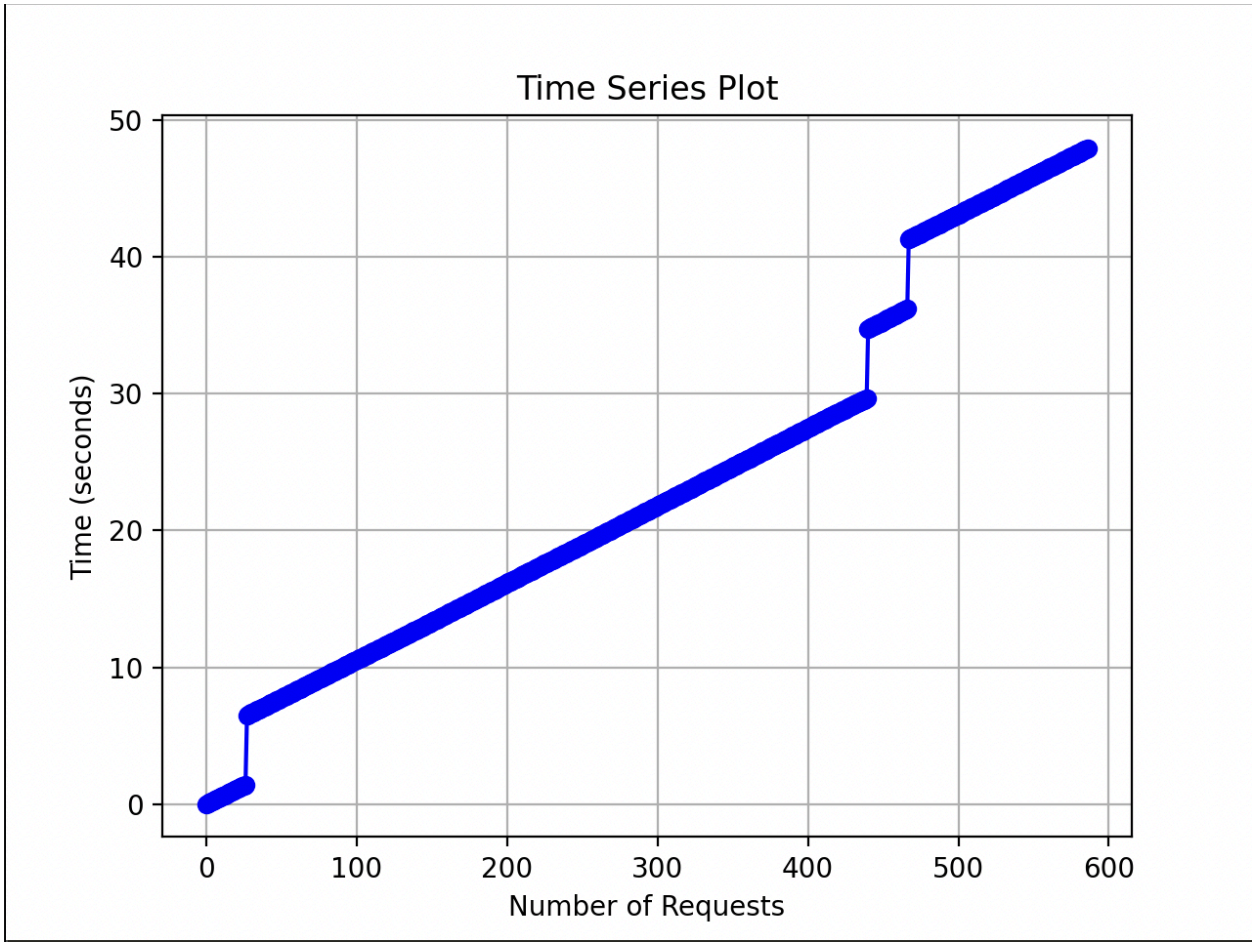


Figure 2: Response time vs Number of requests

The following graph illustrates the impact of leader elections on the program's ability to process requests. The vertical spikes in the graph represent the time taken for the elections to occur. During these periods, no buy requests could be processed, leading to an increase in the overall processing time. In this particular example, three elections took place, and each election caused a temporary halt in request processing, contributing to the observed delays.

7. Additional Test Cases We Ran

Network Setup	Why?
1 Seller → Leader → 1 Buyer	This test case was the happy path test that the election system was taking place properly. And in addition validate that the buyer and seller could both interact with the leader and perform their respective duties.
2 Seller → Leader → 1 Buyer	This test validates several aspects of the program:

One inactive node	1. Validates that only active nodes would be included in the election progress. 2. Validated that concurrent buy requests can be handled from the leader. 3. This test also validated the fall-sick function that would remove a leader node from service.
1 Leader	This test validates that the only node in the network would default to being the leader.
1 Fish Buyer → Leader → 1 Boar Seller	This test case asserted that the Buyer would not find any item for purchase.

Outside of these functional tests the team ran the code that randomized the Peers, Buyers, Sellers, and their roles. This simulation was verified with logs and tested the restocking functionality and the overall system.

8. Conclusion

The peer to peer network system demonstrates how decentralized communication can be used to facilitate product searches in a distributed environment. The use of multi-hop message passing allows buyers and sellers to interact without a central server, and the system scales well under moderate concurrent load. However, as the number of concurrent clients increases, response times rise due to network congestion. Varying the number of neighbors shows a clear tradeoff between network connectivity and response time.