Lab 2 Report
Aylin Elmali and Andreas Gerasimow

Program Design

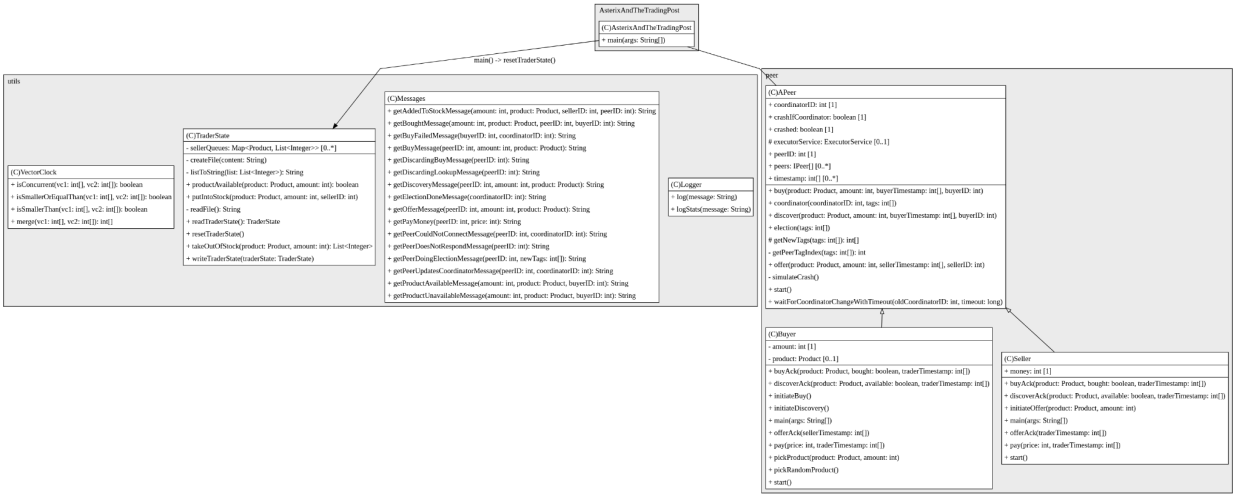
The main components of the program are:

- The **IPeer** interface, which defines the core operations that each peer (whether buyer, seller, or trader) must implement. These include *start()*, which initializes the peer, *getPeerID()* to return a unique ID of the peer, *election()* to send an election message to the next peer, *coordinator()* to update the next peer with the ID of the new coordinator, *discover()* to handle a request from a buyer to check product availability, *discoverAck()* to acknowledge the discover message, *buy()* to handle a buy request from a buyer, *buyAck()* to acknowledge the buy message, *offer()* to handle an offer request from a seller to add products to the market, *offerAck()* to acknowledge the offer message, and *pay()* for the trader to handle payment to the seller
- The functional interface **ElectionHook** used for custom logic during election tasks
- The abstract base class **APeer** that implements common behavior for all peers. It handles peer initialization and network registration, leader election through the *election* and *coordinator* methods (for leader election we chose to implement Ring-based Election), crash simulation using *simulateCrash*, synchronization of the clock (for assignment we chose to implement vector clocks), thread management using *executorService* and *ScheduledExecutorService,* and the operations for trading through the coordinator, using *discover, buy,* and *offer.*
- The **Buyer** class which extends **APeer**, represents the buyer which initiates discovery of product availability and purchases products. It maintains a vector clock to manage logical ordering. It also handles network faults (mainly coordinator crashes).
- The **Seller** class, which extends **APeer**, represents the seller which periodically offers products to the trader to sell. The trader then handles transactions on its behalf. When a sale is made, the seller logs its earnings using the *money* attribute. It maintains a vector clock to manage logical ordering. The Peers use RMI to communicate with the coordinator, and each peer is registered in the RMI registry which allows them to locate the coordinator and neighboring peers.

The utils folder has some additional classes, including:

- **Logger**, which is a class for the peers to print to the console as well as place all output in a log file called “trading_post_log.txt”
- **Messages**, which is a class containing templates for the output messages for the actions of the buyers and sellers and the coordinator
- **TraderState**, which handles the state of the trader, or coordinator. This class manages the inventory of products that sellers have left and tracks product and sellers using a .txt file *trader_state.txt* which allows for recovery if the trader crashes or restarts. It also supports concurrent transactions by updating the stock synchronously.
- **VectorClock**, which is represented as an array of integers where each element tracks the logical time of that peer relative to others. Each peer has its own vector clock. When a peer sends a message it increments its clock, and upon receiving a message it merges its vector clock with the received clock.

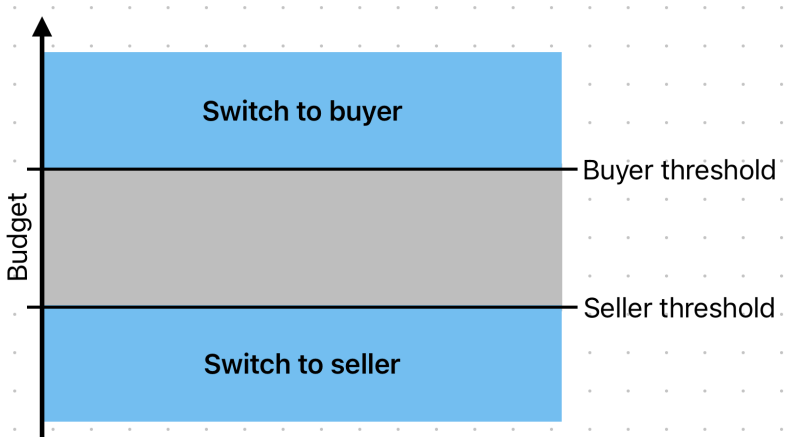
The main start class *AsterixAndTheTradingPost* uses a *ProcessBuilder* to create a Java process for each peer. The peer process's output is redirected to the main process's console and a new thread reads from the peer process's InputStream and writes to System.out.



We chose to implement ring leader elections as this protocol is naturally designed to support concurrent elections in a distributed system. We also chose to use Vector clocks over Lamport clocks to avoid ambiguity, as Lamport clocks only partially order events whereas vector clocks provide a more expressive representation and can explicitly identify concurrent events. The tradeoff is that this requires higher memory and processing overhead. As our project is in Java we opted to use Java RMIs as RMI is part of the Java Standard Library, requiring no external dependencies. The downside is that they are Java-specific, which was fine for our purposes but if we wanted to expand this we might consider using a more modern and scalable protocol.

Future Improvements and Extensions

In our current program peers are assigned to buyer and seller roles, which may not reflect real-world scenarios where roles would need to change dynamically. One possible improvement could be to allow peers to dynamically switch between being a buyer or seller based on market conditions. This might be achieved by combining the logic of the Seller.java and Buyer.java classes or by utilizing the decorator pattern. We may also define conditions to trigger a switch, such as a buyer switching to seller after a certain number of purchases or a seller becoming a buyer after selling all items. Another idea would be to implement buyer budgets. For example, if the budget of a peer falls below a specific threshold, the buyer switches to the seller role and sells items until the budget exceeds a threshold. After that the peer switches to the buyer role



again. We could also enhance the logging by upgrading it from a text file to either JSON or XML and including additional metadata for real-time analytics to monitor the status of the market, e.g. the items available at the trader.

How to run it

Download the source code. There is a README.md file in the root directory that explains how to run it for Unix and Windows systems.

This project uses Java and the Gradle build tool. Therefore, you need Java to run the program. I use the following JDK on my machine:

```
openjdk version "17.0.11" 2024-04-16 LTS
OpenJDK Runtime Environment Corretto-17.0.11.9.1 (build 17.0.11+9-LTS)
OpenJDK 64-Bit Server VM Corretto-17.0.11.9.1 (build 17.0.11+9-LTS,
mixed mode, sharing)
```

To see your Java version, run `java -version`.

Unix-based system:

1. First `cd` into the project folder.
2. Run `./gradlew build` to generate the .jar file. You will see a .jar file in `./build/libs`. **IMPORTANT: Always run this command when changing the source code!**
3. Execute the jar file with `java -jar <path_to_jar_file> <number_of_peers>`.
    a. `<path_to_jar_file>`: Path to the .jar file.
    b. `<number_of_peers>`: The number of peers in the system.

Here is an example of the last step:

```
java -jar ./build/libs/AsterixAndTheTradingPost-1.0-SNAPSHOT.jar 3
```

Windows-based system:

1. First `cd` into the project folder.
2. Run `gradlew.bat build` to generate the .jar file. You will see a .jar file in `\build\libs`. **IMPORTANT: Always run this command when changing the source code!**
3. Execute the jar file with `java -jar <path_to_jar_file> <number_of_peers>`.
    a. `<path_to_jar_file>`: Path to the .jar file. This should be the full path, starting from C:\Users\...
    b. `<number_of_peers>`: The number of peers in the system.

Tests and Known Issues

Under the test folder in AsterixAndTheTradingPost project, we wrote a series of tests to cover various behaviors in the program. When building the Gradle project (as explained in “How to run it”), all tests will run automatically before the .jar file is created.

1. In the CommunicationTest.java class, we tested the communication flow for the *buy* and *offer* methods. We ensured that the timestamps have correct values and that covered various test cases in which the methods might behave differently.
2. In the TraderElectionTest.java class, we tested the leader election. We covered test cases such as concurrent leader elections, reelections, and timeout behavior.
3. The trader state is tested in the TraderStateTest.java class. We tested the methods of the TraderState.java class and the reading and writing to the file.
4. In the VectorClockTest.java class, we tested if concurrent and non-concurrent timestamps are classified correctly. Additionally, we tested the merging of two timestamps.

We only wrote test cases for the individual peer to peer communication, not for the entire network in general, since it’s multithreaded, distributed, and non-deterministic. For example, we didn’t write test cases for generating the network and for deadlock prevention. Writing test cases for this behavior is too complicated. The network runs as expected, even for higher numbers of peers as long as the computer can handle the processes. Issues are currently unknown.

Performance Results

We measured the average response times of 1000 buy requests in three different scenarios. By response time we mean the time between the buy request and the corresponding buy acknowledgement.

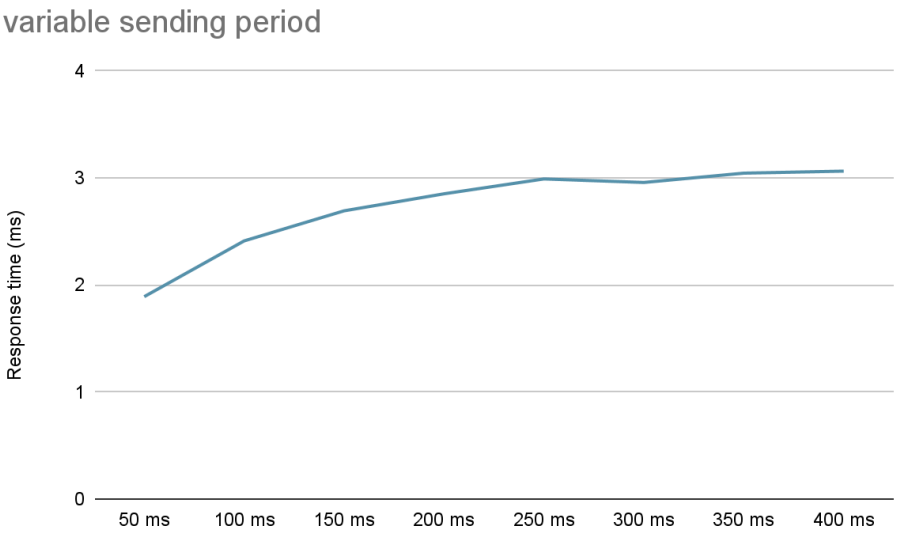
1. The first experiment consisted of six peers and a trader who didn't crash. The sending period of the buyers and sellers was variable. For every sending period between 50 ms and 400 ms with steps of 50 ms, we measured the response time for 1000 successful buy requests.
2. In the second experiment, we measured the average response time of 1000 successful buy requests for a variable number of peers. The sending period of the buyers and sellers was fixed at 500 ms and the trader never crashed.
3. The third experiment had a fixed number of 12 peers and a sending period of 500 ms. The peer with the highest ID (the trader) had variable crash/running durations. For example, the peer would run for 500 ms before it crashes for 500 ms.

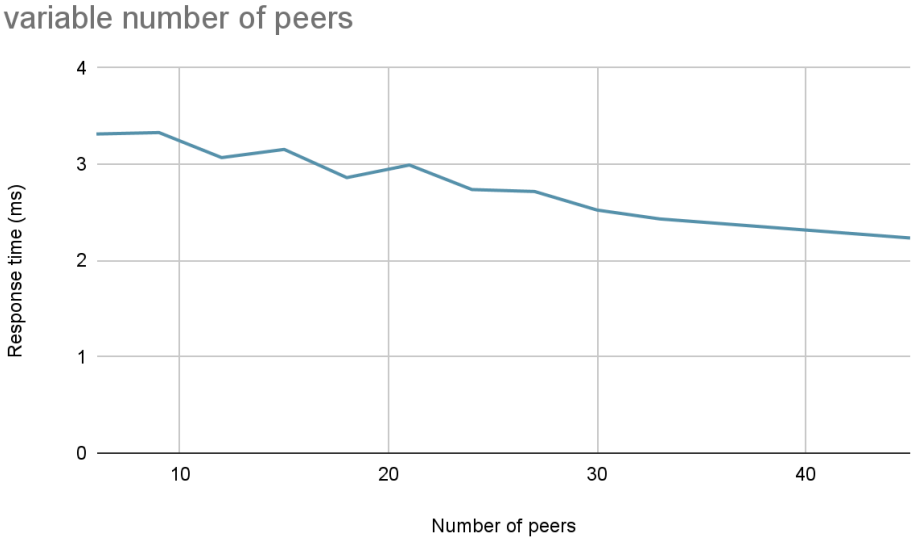
In general, the results were quite unexpected. A smaller sending period, a higher number of peers, and a lower crash duration correlated with a lower response time, even though you would expect the opposite. We thought that this was an error in the time measuring technique. At first, we implemented the time measuring by setting a timestamp *long startingTime = System.currentTimeMillis()* before the buy request. After the buy acknowledgement, we calculated how much time had passed. However, this could lead to issues if another thread updates the *startingTime* before the buy acknowledgement arrives, which might explain the unexpected results. To mitigate this issue, we instead passed the value of the *startingTime* to the trader's buy-RPC and after the buy-RPC call has finished, the trader forwards the same value to the buy acknowledgement. Doing so, the timestamp can never be overwritten. However, we still got the same results as before.

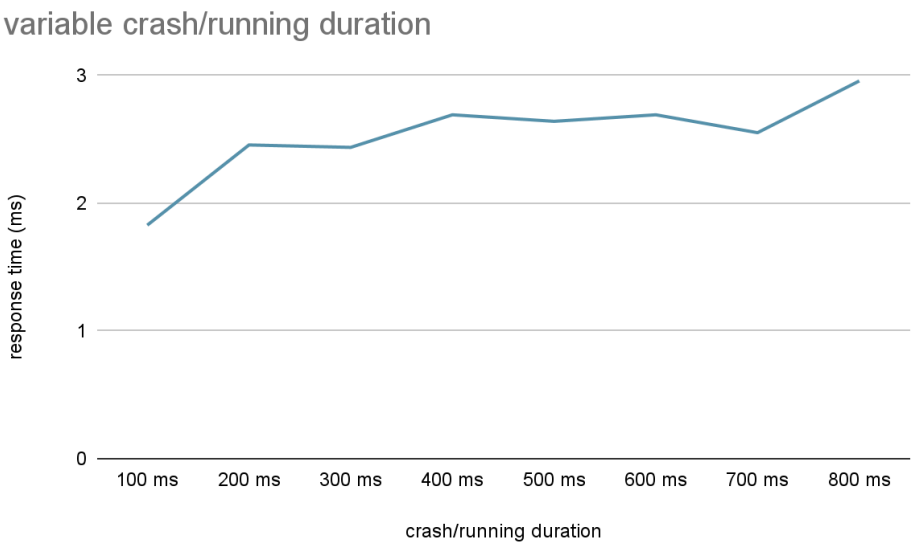
We also noticed that the first responses take longer and then gradually decrease. For example, we conducted an experiment in which we measured the average response time of the first 100 buy requests, then the next 100 buy requests and so on until we reached 1000 requests. The sending period was 400 ms and there were no crashes. We get the following response times:

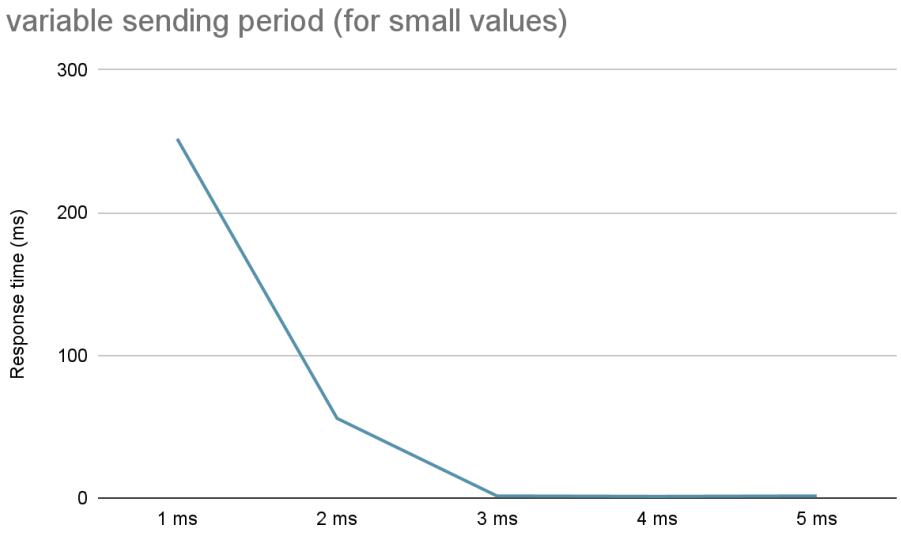
Requests	1 -100	101 -200	201 -300	301 -400	401 -500	501 -600	601 -700	701 -800	801 -900	901 -1000
Resp. time	4.57 ms	3.80 ms	3.21 ms	3.05 ms	2.54 ms	2.82 ms	2.94 ms	2.51 ms	2.74 ms	2.46 ms

We suspect that this has something to do with how Java works internally. The more RMI calls are executed by the processes, the more resources the JVM allocates to them.To strengthen this argument, we repeated the first experiment with very small sending periods between 1 ms and 5 ms. You can see that for sending periods smaller than 3 ms, the response time grows very fast.






variable crash/running duration
response time (ms)
3
2
1
0
100 ms
200 ms
300 ms
400 ms
500 ms
600 ms
700 ms
800 ms
crash/running duration


variable sending period (for small values)
Response time (ms)
300
200
100
0
1 ms
2 ms
3 ms
4 ms
5 ms