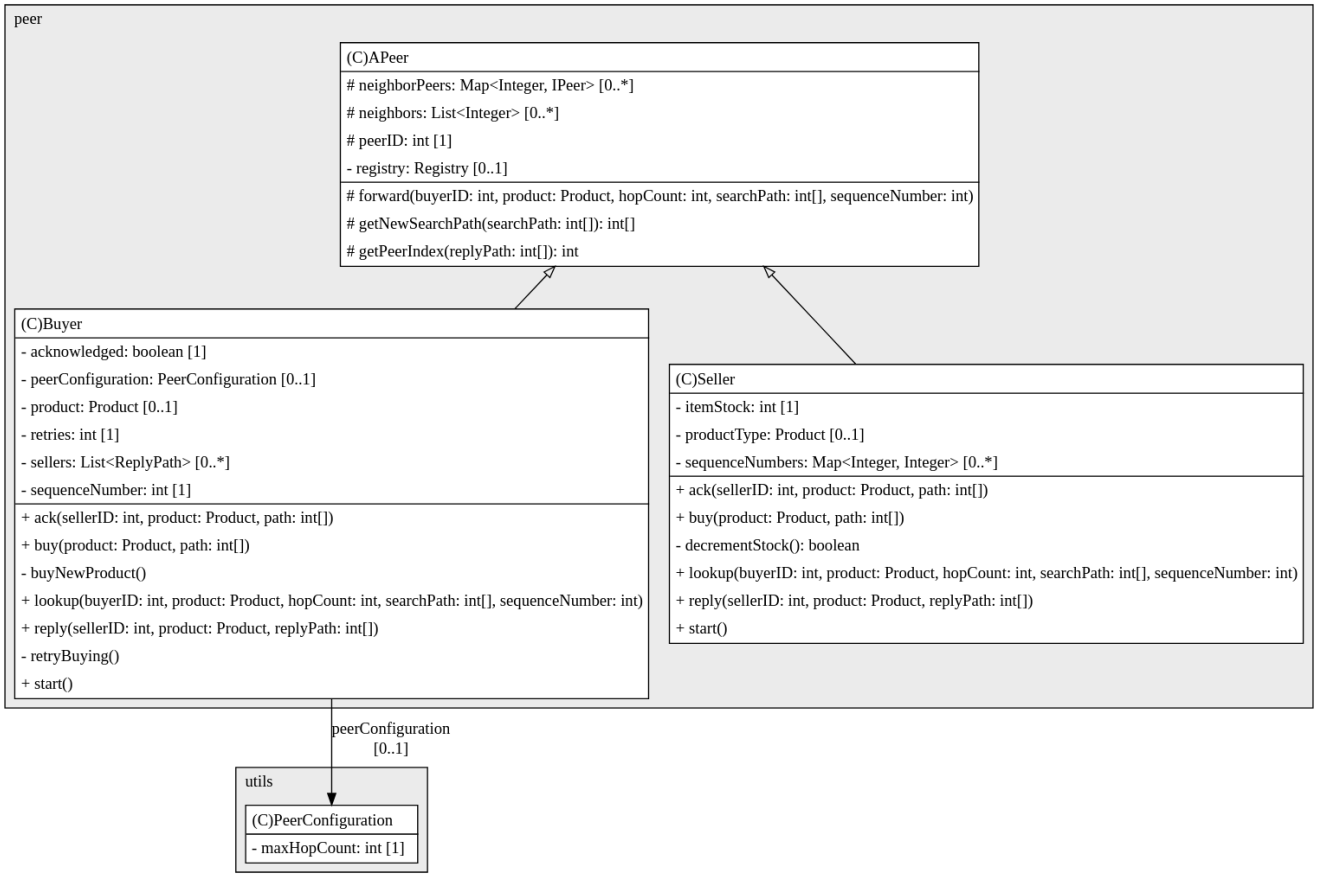
Lab 1 Report
Aylin Elmali and Andreas Gerasimow

Program Design

The main components of the program are:

- The **IPeer** interface, which defines the core operations that each peer (whether buyer or seller) must implement. These include *start()*, which initializes the peer, *lookup()* to handle lookup requests from buyers, *reply()* to respond to buyers, *buy()* to process a buy request from a buyer, *getPeerID()* to return a unique ID of the peer, and *getNeighbors()* which retrieves a map of direct neighbors connected to the peer.
- The abstract base class **APeer** that implements common behavior for all peers, which handles message forwarding and tracking neighbor peers.
- The **Buyer** class which extends **APeer**, represents the buyer which initiates lookups and makes purchases. Buyers use a scheduled task via *ScheduledExecutorService* to periodically initiate product searches or retries.
- The **Seller** class which extends **APeer**, represents the seller which responds to buyer lookup requests for products. The peers use RMI to communicate, and each peer is registered in the RMI registry which allows other peers to look them up and send a message.



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 “bazaar_log.txt”
- **Messages**, which is a class containing templates for the output messages for the actions of the buyers and sellers
- **PeerConfiguration**, a class for getting and setting the maxHopCount

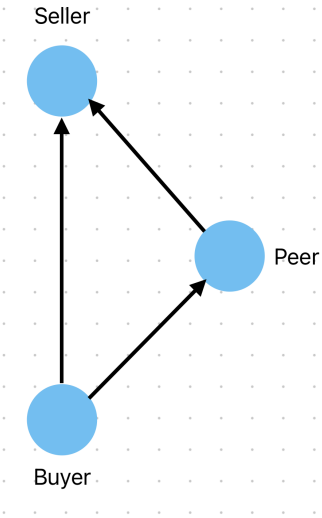
The **Buyer’s** action flow is as follows:

1. The buyer initiates a product search *lookup()* by forwarding the request to its neighbors
2. Peers forward the lookup message until it reaches a seller or reaches the maximum hop count
3. If a seller has the requested product, it sends a *reply()* back to the buyer
4. The buyer collects replies from sellers and and sends a *buy()* request to one of them
5. If the buy fails due to stock depletion, the buyer retries or starts searching for a new product

The **Seller’s** action flow is as follows:

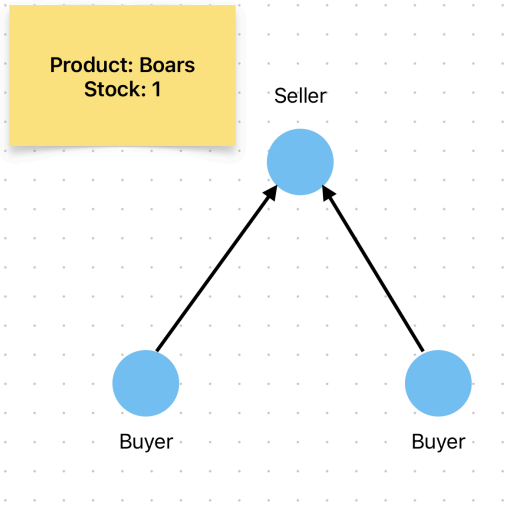
1. The seller receives a lookup message and checks if it has stock of the requested product
2. If it does, it sends a reply back to the buyer through the peer network
3. When the seller receives a *buy()*, it processes it, updates the stock using a synchronized method *decrementStock()*, and sends an *ack()*
4. If stock is depleted, the seller selects a new random product

We accounted for certain edge cases, one being where a seller receives the same buy request multiple times from different paths:



We handled this case by implementing a unique sequence number for each buy request to prevent duplicate messages from being processed twice.

The second case we handled was when a seller only has one item left in stock and receives two buy requests at once:



For this case, we implemented a three retry timeout in *retryBuying()*. If the buyer is not able to successfully make a purchase after three tries it starts searching for a new product in *buyNewProduct().* One possible improvement or extension if we were to continue developing this could be to introduce an exponential backoff mechanism to increase the wait time between retries to mitigate network congestion. Another could be to implement handling of network partitioning, as it's possible some peers could become isolated due to neighbor failures. If given more time we could consider adding a mechanism for buyers or sellers to detect when they are isolated and signal an error or retry with different peers.

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`.
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/AsterixAndTheBazaar-1.0-SNAPSHOT.jar 2
```

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`.
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 **AsterixAndTheBazaarTest.java** we wrote a series of tests for different aspects of 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. We first tested the communication flow of all methods: *lookup*, *reply*, *buy*, and *ack*. The methods in the test cases are forwarded by multiple buyers and sellers via RMI to assure that all peers behave correctly. Additionally, the tests assert that all values of the messages arrive correctly at the receiving peer.
2. Next, we tested the behavior when the lookup method arrives at multiple sellers. We asserted that the lookup arrives at all sellers and they send a reply to the correct buyer. We also asserted that the buyer receives the correct number of replies and that the values of the replies are correct.
3. Finally, we tested the behavior when a seller receives exactly the same lookup message from the same buyer. This can happen when both peers are connected through multiple paths. We asserted that the first message will be processed and that the seller sends a reply afterwards. We also asserted that the second lookup will be discarded. Additionally, we asserted that another distinct lookup message from the same buyer won’t be discarded.

We only wrote test cases for the individual peer communication, not for the entire network in general. For example, we didn’t write test cases for generating the network and for deadlock prevention. Our test cases don’t cover the entire network, since it’s multithreaded, distributed, and non-deterministic. Writing test cases for this is too complicated. The network runs as expected, except if the number of peers exceeds around 50. Then the following error occurs:

```
Exception in thread "Thread-26" Exception in thread "Thread-29"
Exception in thread "Thread-10" Exception in thread "Thread-78"
Exception in thread "Thread-63" Exception in thread "Thread-22"
java.lang.RuntimeException: java.rmi.ConnectIOException: error during
JRMP connection establishment; nested exception is:
      java.net.SocketException: Connection reset
```

We think this error is caused either because the Java VM uses too much memory or because of specific socket timeout settings. However, we tried to increase the memory and the socket timeouts but the error still persists.

Performance Results

We tested the performance of the peers by analyzing the time difference between the lookup and the reply. For this, we created a network of 50 peers. We calculated the average response time per 1000 replies for a variable amount of connections per peer. We found that a higher number of connections per peer correlates with a lower average response time, as shown in the graph below.