Concurrent and Event-Based Programming

<u>Bursa</u>

Team: Bits_please
Members: - Sentaliu Georgiana
- Stefaniga Beniamin
- Szatmari Larisa
- Vlad Catalin-Andrei

Content

1. Description of implementation
2. Synchronization mechanisms
3. Description of concurrency problems found
4. Roles and contributions of team members
5. Bibliography

1. Description of implementation

This project was written in Java programming language, using Eclipse IDE. It was implemented with “Client-Server” design pattern, using “java.net” sockets.

1.1. Classes

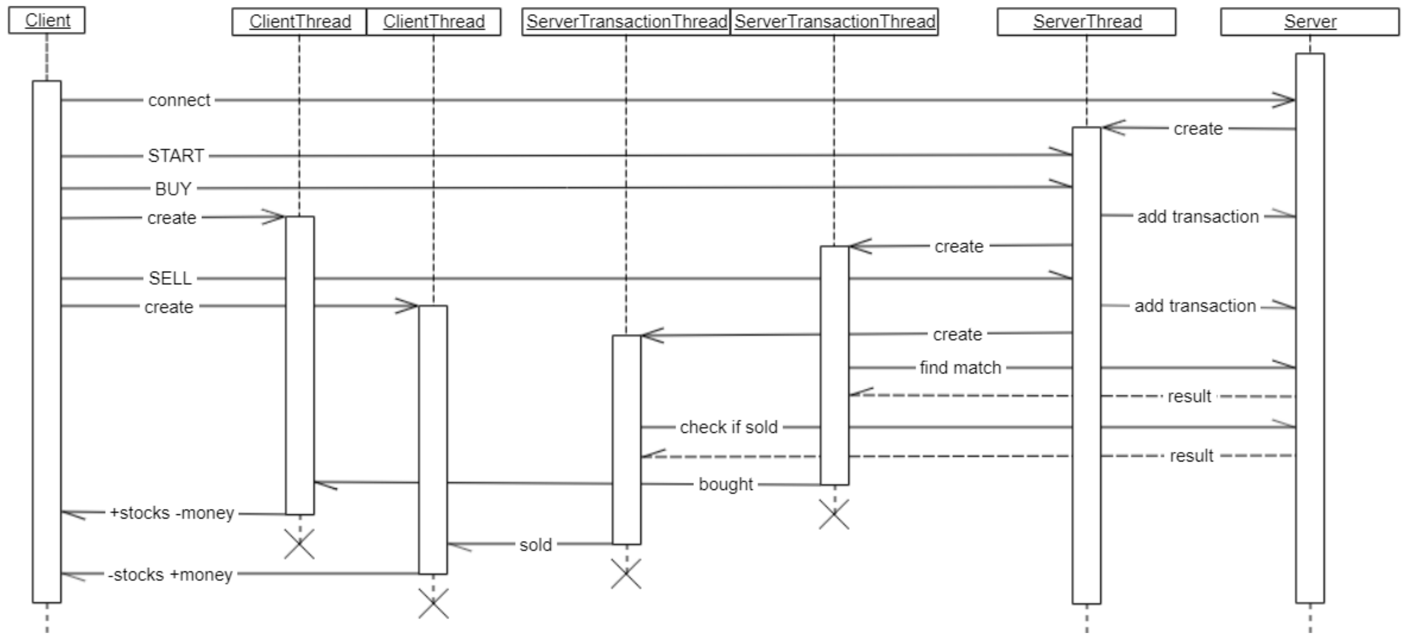
The project is structured, in packages and classes, as follows:

- transaction.package
    - ITransaction – Interface containing all the method declarations used in a transaction
    - Offer – Class that represents an offer made by a client(selling stocks)
    - Request – Class that represents a request made by a client(buying stocks)
- server.package
    - Server – Class that represents the main server thread. For every client that tries to connect it creates a new ServerThread object. It also holds and modifies the transaction pool in a <u>synchronized</u> way.
    - ServerThread – Class that handles a single Server-Client connection. It waits for transactions from the client and for each it creates a ServerTransactionThread.
    - ServerTransactionThread – Class that handles a transaction. It tries to find a match for the request/offer transaction and then it sends a result to the client. It uses the Server object lock to <u>synchronize</u> search.
- client.package
    - Client – Class representing a client. It <u>synchronizes</u> changes to its attributes and creates a ClientThread for every transaction it sends to the server.
    - ClientThread – Class that waits for result from the server and modifies the client attributes accordingly.

1.2. Protocol description

The protocol is described below, followed by a sequence diagram:

- Client connects to the server
- Server creates ServerThread to handle the connection with that Client
- Client starts the communication with the ServerThread and sends a request(or more)
- For every transaction received, ServertThread creates a ServerTransactionThread to handle that specific transaction
- Likewise, Client starts a ClientThread for every request he sent, to wait for that transaction's result from the ServerTransactionThread
- ServerTransactionThread searches a match for that specific transaction and sends a result to the ClientThread
- ClientThread waits for a result, and when it's received, it changes Client attributes accordingly.



2. Synchronization mechanism

The only primitive used in this project is the monitor, which in java is masked by the keyword synchronized. Each java object has a monitor lock which is automatically acquired when entering a synchronized method.

- Server class holds a pool of transactions and all the methods that can modify or read the pool, so the object's monitor lock was used to synchronize accesses to the pool:

```
public class Server implements Runnable
{
	// Transaction pool
	private HashMap<String, ITransaction> transactions_;
	// List containing all previously completed transactions
	private ArrayList<Pair<String, String>> transactionHistory_;
	...

	public synchronized ArrayList<String> getTransactionHistory()
	{
		/* Method that returns a copy of the transaction history list
		 * Synchronization is needed because if the list is being modified
		 * while being copied, incomplete and irrelevant data will be sent
		 * further.
		 */
		...
	}

	public synchronized ArrayList<String> getTransactions()
	{
		/* Method that returns a copy of all the pending transactions.
		 * Synchronization is needed for the same reason as the previous method
		 * In the best case we copy irrelevant data, but it can also result in
		 * a program crash because we can try to access a transaction that was
		 * deleted by another thread.
		 */
		...
	}

	public synchronized void addTransaction(ITransaction t)
	{
		/* Method that adds a transaction to the transaction pool
		 * Synchronization is needed because two threads can try to add
		 * transactions at the same time. Undefined behavior.
		 * It can lead to crashes.
		 */
		...

	}
	...
}
```

- ServerTransactionThread class uses Server’s object monitor lock when searching for a match, to make the operation atomic.

```
public class ServerTransactionThread extends Thread
{
        ...
        public void run()
        {
                ...
                synchronized(server_)
                {
                        Offer o = server_.find((Request) transaction_);
                        if( o !=null )
                        {
                                String write = "YOUBOUGHT " + "...";
                                outputString.println(write);
                        }
                }
                /* This block searches for an offer that matches the given request.
                 * It needs to be synchronized because the operation must be atomic
                 * in order to find an offer, delete it from pool and send a message to
                 * the client.
                 */
                ...
        }
        ...
}
```

- Client class uses intrinsic lock to synchronize its attribute modifications

```
public class Client implements Runnable
{

        private int numberOfStocks_;
        private int money_;
        private boolean buyCompleted_;
        private boolean sellCompleted_;
        ...

        public synchronized void setSellCompleted(boolean b){...}
        public synchronized boolean getSellCompleted(){...}
        public synchronized void setBuyCompleted(boolean b){...}
        public synchronized boolean getBuyCompleted(){...}
        /* Synchronized getters and setters for two attributes
         * Need synchronization because they can be read and set at the same time by
         * different threads. It took us quite a bit of time to figure this out :)
         */
        public boolean sold(Offer myOffer)
        {
                ...
                synchronized(this)
                {
                        numberOfStocks_ = numberOfStocks_ - stocksSold;
                        money_ = money_ + stocksSold*price;
                }
                ...
        }

        public boolean bought(Offer offer)
        {
                ...
                synchronized(this)
                {
                        numberOfStocks_ = numberOfStocks_ + stocksBought;
                        money_ = money_ - stocksBought*price;
                }
                ...
        }
        /* Methods that modify stocks and money of a client
         * Need synchronization so that modifications happen one after the other
         * Undefined behavior without synchronization.
         */
        ...
}
```

3. Description of concurrency problems found

We found the following problems during development, which we avoided using monitors:

- Irrelevant data read: When Server::getTransactionHistory() was called without synchronization, sometimes it could read half written data which was irrelevant.
- Random crashes: When Server::getCurrentTransactions() was called without synchronization, sometimes it could try to read a memory location of a transaction that was already deleted, which results in a crash.
- Thread block: Perhaps the most annoying problem we faced (even worse than crashes because there we at least had the stack trace) was when we tried to read “boolean buyCompleted_” and “boolean sellCompleted_” from Client class. The client was supposed to wait until both buy and sell transactions were finished so he can send some more transactions. Unfortunately, in a different thread we changed these attributes and the client would try to read the values while being written and the thread would just block. It took us a long time to find this problem because there were not prints, no exceptions or crashes and we couldn’t pinpoint the source of the problem. We solved it by changing and reading the attributes with the help of some synchronized methods.

4. Roles and contributions of team members

We started with a sketch of a sequential diagram and decided the number of threads needed for a single Client-Server transaction. After defining the protocol, we divided the work for each member as following:

- Sentaliu Georgiana: ServerTransactionThread class and ServerThread class.
- Stefaniga Beniamin: ClientThread class and ServerTransactionThread class.
- Szatmari Larisa: ServerThread class and Server class.
- Vlad Catalin-Andrei: Client class and ClientThread class.

The ITransaction, Offer and Request classes were done first by all team members.
Classes that interconnect with other classes were handled by two people in order to assure a good understanding and good communication between them.
Code refactor was done at the end.

5. Bibliography

PCBE Curs 2019 – Dan Cosma
Java SE 8
Socket Programming in Java – Souradeep Barua

Git repository can be found here: https://github.com/catalinvlad192/PCBE/tree/master/PCBE_Bursa