University of Essex  
CE303 Advanced Programming

Client-Server System Report

Word count = 1062   
(without title page, list of contents, implementation table and appendices)

Prepared by  
Michal Borowski  
1904535  
mb19424@essex.ac.uk

4th December 2020

Table of Contents

[Implemented functionality 3](#_Toc57903684)

[Protocol 4](#_Toc57903685)

[Opening sequence 4](#_Toc57903686)

[Client commands 5](#_Toc57903687)

[Recognition of client or server going down 5](#_Toc57903688)

[Server event-triggered commands 6](#_Toc57903689)

[Client Threads 7](#_Toc57903690)

[Server Threads 7](#_Toc57903691)

[Project review 8](#_Toc57903692)

[Appendix A - Justification of design choices 9](#_Toc57903693)

# Implemented functionality

|  |  |  |
| --- | --- | --- |
| **Function** | **C#** | **Java** |
| Client establishes a connection with the server | Yes | Yes |
| Client is assigned a unique ID when joining the market | Yes | Yes |
| Client displays up-to-date information about the market state | Yes | Yes |
| Client allows passing the stock to another player | Yes | Yes |
| Server manages multiple client connections | Yes | Yes |
| Server accepts new connections while traders are exchanging stock among themselves | Yes | Yes |
| Server correctly handles clients leaving the market | Yes | Yes |
| Client is compatible with the server in the other language | Yes | Yes |
| **Additional tasks:** | | |
| Client GUI | Yes | Yes |
| Server GUI | Yes | Yes |
| Server restarts are correctly implemented | Yes | Yes |
| Unit tests | Yes | Yes |

# Protocol

Every message of the protocol is a separate line. I used orange to highlight raw messages, and blue to highlight their format.

## Opening sequence

Figure . Opening sequence, communication just after client connects. Client proves identity and gets the state of the market.

1. Client sends unique token to the server.

2. Server responds with 3 semicolon separated values (see Figure 1).

Received ID depends on the token, the server searches for it in a “token:ID” dictionary of previously connected clients (saved/loaded from a file, see Figure 2). If key with specified token is found then the corresponding ID value is included in response, if not, then a new ID is created and included.

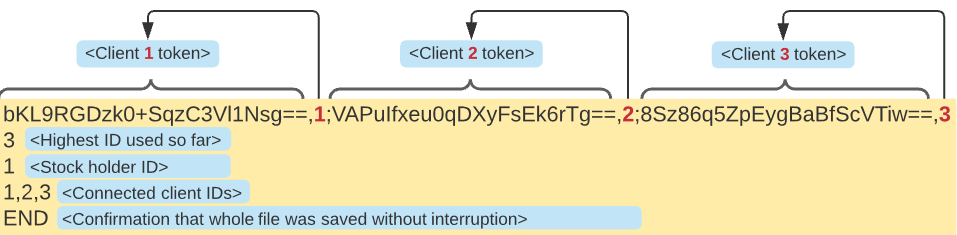


Figure 2. Contents of "%appdata%/CE303\_server\_state.txt" file. Orange color marks the actual content.

3. If server restarts, client uses previously generated token to claim/get its old ID back.

After opening sequence, client and server, send and receive independently. This way clients can get immediate responses about market events without requesting it (see Figure 3 where clients get notified about stock transfer).

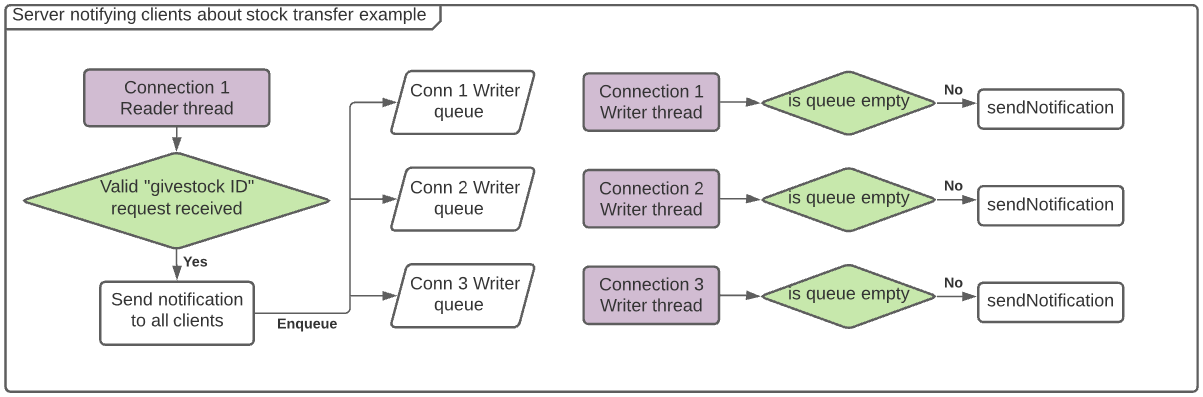


Figure 3. How the server notifies clients about events such as stock transfer (all “Connection” threads are part of the server).

## Client commands

The only command clients can send to the server is:

givestock 5  
givestock <receiver ID >

In case of success, **all clients receive** **response** of which the format is always:

givestock success <owner ID> <receiver ID>

In case of failure, only the owner receives response of which format is always:

givestock fail <reason string>

Table 1 shows possible server responses to it.

|  |  |
| --- | --- |
| **Circumstances** | **Response from server** |
| Trader with supplied trader ID exists in the market. | givestock success 1 5 |
| Non-integer, or no number at all is supplied as receiver ID. | givestock fail Wrong format used for 'givestock id' |
| Supplied receiver ID is not in the market. | givestock fail Supplied trader id (5) does not exist. |
| The client who sent the command does not have the stock. | givestock fail You are not in possession of a stock. |

Table . Possible server responses to "givestock <receiver ID>" command.

## Recognition of client or server going down

Every second, both, client and server send a line containing a dot. This is done in asynchronous manner, and no response is ever sent back to these messages. The purpose of it is to check that a program at the other end went down and the connection is broken.

## Server event-triggered commands

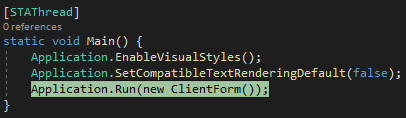
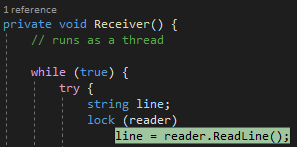
| **Server event** | **Message from server to all clients** |
| --- | --- |
| Client joins | connected 5  connected <trader ID>  If server restarts after going down while having connected clients, and the newly connected client is one of them, then additional argument (“reconnected”) is appended to the command above, such as:  connected 5 reconnected  connected <trader ID> reconnected  This way the clients won’t display “Trader <ID> joined.” which would be not elegant. |
| Client leaves | disconnected 5  disconnected <trader ID>  If the leaving client was a stock holder, and there is at least 1 remaining client, then the following command is sent before “disconnected <trader ID>”:  givestock left 5 1  givestock left <trader ID who left> <receiver ID>  Receiver ID happens to be the one who joined first.  If not a single client is connected back after server restart, the stock is released from the owner anyway and is given to the first new client. |
| At least 1 client does not reconnect within 5 seconds after server restart. | didnt\_reconnect 1,2,3  didnt\_reconnect <csv list of trader IDs that didn’t reconnect>  If the client who didn’t reconnect was a stock holder, and there is at least 1 remaining client, then the following command is sent before “didnt\_reconnect <ID list>”:  givestock didnt\_reconnect 3 5  givestock didnt\_reconnect <owner ID> <receiver ID>  Receiver ID happens to be the one who joined first.  If not a single client is connected back after server restart, the stock is released from the owner anyway and is given to the first new client. |

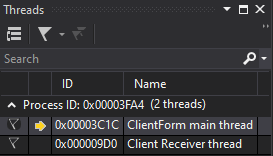
Table . Messages sent from server to all Clients after certain events.

# Client Threads

The client runs 2 threads named:

|  |  |
| --- | --- |
| **ClientForm main thread**  It handles GUI, requests stock transfer and checks if server is down. | **Client Receiver thread**  It establishes and maintains connection, which involves completing “opening sequence”. It receives messages from server and requests GUI updates. |





Both threads described above are terminated when the client program is closed or its process is killed.

# Server Threads

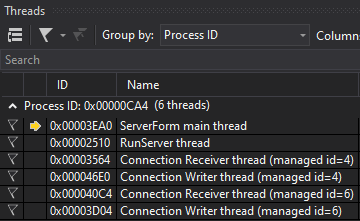
The server constantly runs 2 key threads:

Figure . Example threads of a server with 2 connected clients.

**ServerForm main thread** is responsible for handling GUI.

Client 1

**RunServer thread** is responsible for accepting new tcp connections. For each connection is creates an instance of “Connection” class which spawns 2 threads, 1 for receiving and 1 for writing.

Client 2

Both of these key threads are terminated only when the program terminates.

**Connection Receiver thread** is responsible for getting information from the client, it also recognizes that the client disconnected in regular way as shown in Figure 5.

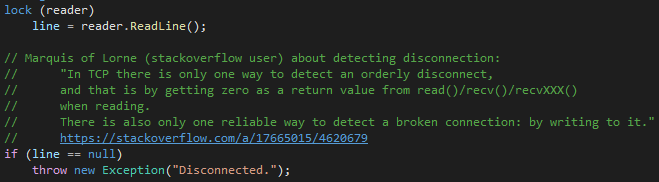


Figure 5. How server recognizes that client disconnected in regular way.

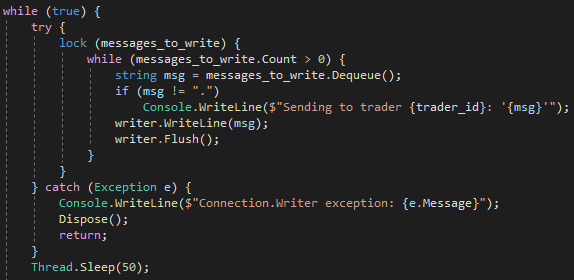
**Connection Writer thread** is responsible for sending information to client. It does that by removing strings from “messages\_to\_write” queue until it’s empty. It also recognizes that the server connection is broken.

Figure . Try/catch statement recognizing broken connection.

Both, Writer and Receiver threads are terminated when the client disconnects (regardless if the connection is broken or it’s orderly disconnection).

# Project review

Despite temptation to use workarounds, I managed to implement most of the key features the best way I could think of, including:

* concise protocol
* efficient communication (sending/receiving independently from separate threads).
* elegant locking mechanism using system-wide mutex to prevent multiple servers being started at once. In Java it required using WinAPI calls (accomplished with JNA library).
* carefully designed transparent server restart:
  + market state is restored even if the newly started server was written in another language, thanks to using common (%appdata%) storage location. Backup file prevents data loss due to power off while saving.
  + server allows 5 seconds for the original stock owner to reconnect (then it releases he stock)
  + server allows 5 seconds for previously connected clients to reconnect (then it notifies other clients about lack of reconnection)
* almost identical C#/Java implementations

For me, the most difficult part was to manage increased complexity of additional tasks due to bad design choices I made in early stages.

# Appendix A - Justification of design choices

**Why use both, ID and token?**

ID is easy to read, token is hard to guess. ID allows easily distinguishing clients (by the user), token proves identity upon reconnecting.

**Notes about token, ID and opening sequence:**

Token can be any non-empty string not containing space, semicolon and coma. IDs start with 1 and increase by 1 with every new connected client with previously unseen token (server saves and restores the highest ID in a file). If client is the only trader, the stock holder ID is the same as his ID. List of trader IDs contains self ID.

**Why bunch of unintuitive semi-colon separated values is sent at the beginning?**

To provide initial state of the market for the client in the simplest form. This way the further communication includes only the changes that occur (the whole state of the market is not unnecessarily sent again).

**Why client has only one command?**

Because giving stock is the only action clients can undertake in the market. Adding more commands (e.g. requestStockHolderUpdate, requestId, requestAllTraders) that would probably be called only once, would unnecessarily complicate protocol. Using this design, protocol could be extended and still keep clarity.