Table of Contents

Design Overview

The Java Chat Application has been designed by sub-dividing the required components into packages, with each package containing one or more classes. Object-orientated design principles have been adhered to (e.g. the *Single Responsibility Principle*) and common functionality is abstracted into reusable classes to which responsibility is delegated. The final application is ***fully interactive, threaded,*** it ***supports multiple servers, multiple clients, and client-to-client communication over different servers,*** it allows the user to ***configure port/host names at the command line for server and client***, and it ***manages disconnection/reconnection in a safe manner***. The packages and their classes are as follows with important features noted:

runners package: contains the two main client-server programs (with *main()* methods) which are launched by the user to start chat sessions:

- ***runners.ChatClient.java***: composed of *TerminalInputParser* and *RunnableCommandLineOptionMenu* objects (which facilitate interactive user sessions where the user chooses tasks from a pre-built menu of options), and a *Client* object (which manages state and behaviour related to connecting to the server). Contains functionality which allows the user to set the port and hostname to which the client will connect, to connect to said server, to enter a user defined message, to choose a list of clients (or the ChatServer) to send the message to, to forward the message to the chosen clients, print received messages and detailed event logs to the screen, and to disconnect from the server and subsequently reconnect with all previous status/events/messages for the session still available. The *startMonitor* and *monitorEventLog* methods are ***threaded*** and they monitor the *Client EventLog* in order to parse incoming messages (including lists of other *Clients* which are forwarded by the server), and to monitor the Socket connection status (i.e. disconnection events) so that users can reconnect if servers are unavailable are lost.
- ***runners.ChatServer.java:*** the *ChatServer* also has *TerminalInputParser* and *RunnableCommandLineOptionMenu* objects to facilitate interactive user sessions, an *EventLog* (for storing acting upon message/socket events and storing general status), and a list of *MultiClientServer* objects. Each *MultiClientServer* object in the *ChatServer* contains much of the functionality of the actual server and the *ChatServer* wraps that functionality in an interactive way. Contains functionality to set the port of the server, to launch one or more servers on different ports, to enter a chat message and forward it to specific clients or all connected clients, to distribute a list of all connected clients to all other clients to facilitate client-to-client communication, and to safely close all client connections. The *startClientMonitor* and *startMessageRelay* methods are ***threaded*** and they monitor the *MultiClientServer* and *MultiClientServerThread EventLogs* in order to distribute lists of connected clients across the network, and to relay messages between clients.

client package: contains a single *Client* class which manages *Sockets*, *EventLogs*, and reading and writing to/from input/output streams.

- ***client.Client.java***: composed of *Socket* (from java.net.*), *EventLog, StreamReader* and *StreamWriter* objects with "*bean"* methods to used set/get ports/hostnames, lists of other clients, the current connection status, etc. The *connect* and *disconnect* methods are the key methods which attempt to open and close the connection to a *MultiClientServer or MultiClientServerThread*. If a connection is made, a *StreamReader* thread is started which maintains an input stream from the server, the *StreamReader* pushes incoming messages to the *Client EventLog* which is monitored by a thread in the *ChatClient* in order to facilitate live user interaction. When the user sends a message via the *ChatClient* it does so by invoking the *sendMessage* method of *client.Client.java* which in turn delegates responsibility for sending that message to the *StreamWriter* class.

server package*:* contains two classes, *MultiClientServer* (which implements the server socket functionality) and *MultiClientServerThread* (which extends *Thread* and is spawned when a client connects to the server, this thread maintains the *Socket* and communicates with the client):

- ***server.MultiClientServer.java:*** When a *Client* connects to a *MultiClientServer*, the *Client* is assigned its own *MultiClientServerThread* to facilitate ongoing communication. This class contains a range of getter/setter methods for retrieving/updating lists of connected/disconnected *MultiClientServerThread* objects. The *startMultiClientThreadMonitor* method is ***threaded*** and it processes the *EventLogs* of all *MultiClientServerThread* objects in order to monitor their connection status and to facilitate client-server and client-to-client communication.
- ***server.MultiClientServerThread.java:*** is composed of a *Socket*, *StreamReader* and two *EventLog* objects (one for general events, another for message events to forward to other clients). The *startEventMonitor* method is ***threaded*** and it monitors the *MultiClientServerThread EventLog* to keep track of active/inactive connections, and to parse messages from clients which can subsequently be dealt with hierarchically by the *MultiClientServer* and the *ChatServer.*

event package*:* contains three classes – *EventLog*, *Event* (the basic unit of which an *EventLog* is composed), and *EventLogPrinter.* The *Event* and *EventLog* classes are used as logging devices (similar to built-in Java logging packages) rather than printing messages directly to the screen. The *EventLog* details can be printed upon user request. Detailed *EventLogs* for clients and servers can be viewed by a user via built in options in the interactive menus of the *ChatClient* and *ChatServer.*

- ***event.Event.java:*** events are used as the basis of maintaining connection status and relaying messages within and between clients and servers. Each *Event* has a *type*, *time, text* and *serviced* field. The *type* varies depending on the type of event that occurred and is used to distinguish between general status update events, and more specific connection, incoming message, and outgoing message events. The *time* field dictates when an event occurred, and the *text* field is some messages associated with the event (a user message, or an internally generated one). The *serviced* field is boolean – if *serviced* is true, any threaded methods which monitor *EventLogs* throughout the application do not need to act or have already acted on the event and have therefore set the *serviced* field to true. If *serviced* is false it

means that a monitor/thread somewhere in the application must service the event or carry out some action which facilitates client-to-client or client-server communication.

- ***event.EventLog.java:*** an *EventLog* is simply a list of *Events*. Many of the classes described above rely on having their *EventLog* monitored in order to facilitate interactive user sessions. Since *EventLogs* are passed between classes and may be concurrently monitored/altered by different threads, the they are typically accessed from *synchronized* methods throughout the application and concurrent lists are used.
- ***event.EventLogPrinter.java:*** contains basic code to print the *Events* in *EventLogs* stored in client and server classes to the screen in a readable format – it is simple logic but the methods are used in both the client and server side applications.

stream package*:* contains three classes related to input and output streams – *StreamReader, StreamWriter* and *SocketHandler*.

- ***stream.StreamReader.java:*** the basic functionality to retrieve incoming messages is abstracted into this class, and the client and server classes described above are typically composed with their own *StreamReader* instance which awaits input from the Socket and adds the received messages to an *EventLog*. It also contains a *parseMessage* method which is used by other classes to parse received messages from clients/servers. This class extends the *Thread* class – when the *StreamReader* is started, it awaits messages and when it receives one it adds it to an *EventLog.* Whatever object is composed with that *StreamReader* then processes the *Event* via its *Event* monitoring threads.
- ***stream.StreamWriter.java:*** the basic functionality to write outgoing messages is abstracted into this class. *StreamWriter* instances are provided with a *Socket* to write the message to, a message to send, and an optional source and destination address (all communication is through the *Socket*, but to facilitate client-to-client communication the source and destination address may be varied from those associated with the *Socket*). Also contains a method *formatOutgoingMessage* to format messages in a standard format (src=[*] dest=[*] message=[*]) so that *StreamReaders* can parse the message consistently.
- ***stream.SocketHandler.java:*** a trivial class that returns string based information when provided with a *Socket* e.g. addresses for printing to screens.

terminalIO and menu packages*:* These two packages and their associated classes were written and designed entirely by this author as part of a previous project completed during the Advanced Object Orientated Programming module and they were adapted to provide interactive interfaces and command-line parsing functionality for the *ChatServer* and *ChatClient*. For brevity they will not be described here, but the source code is well-documented in the .java files.

Instructions for Using Application

ChatServer

Open a terminal in the *src/* or *bin/* directory (depending on whether you have compiled the source code yourself or using an IDE like eclipse) and launch the application via:

- *java runners.ChatServer*

You will be presented with an option menu as follows:

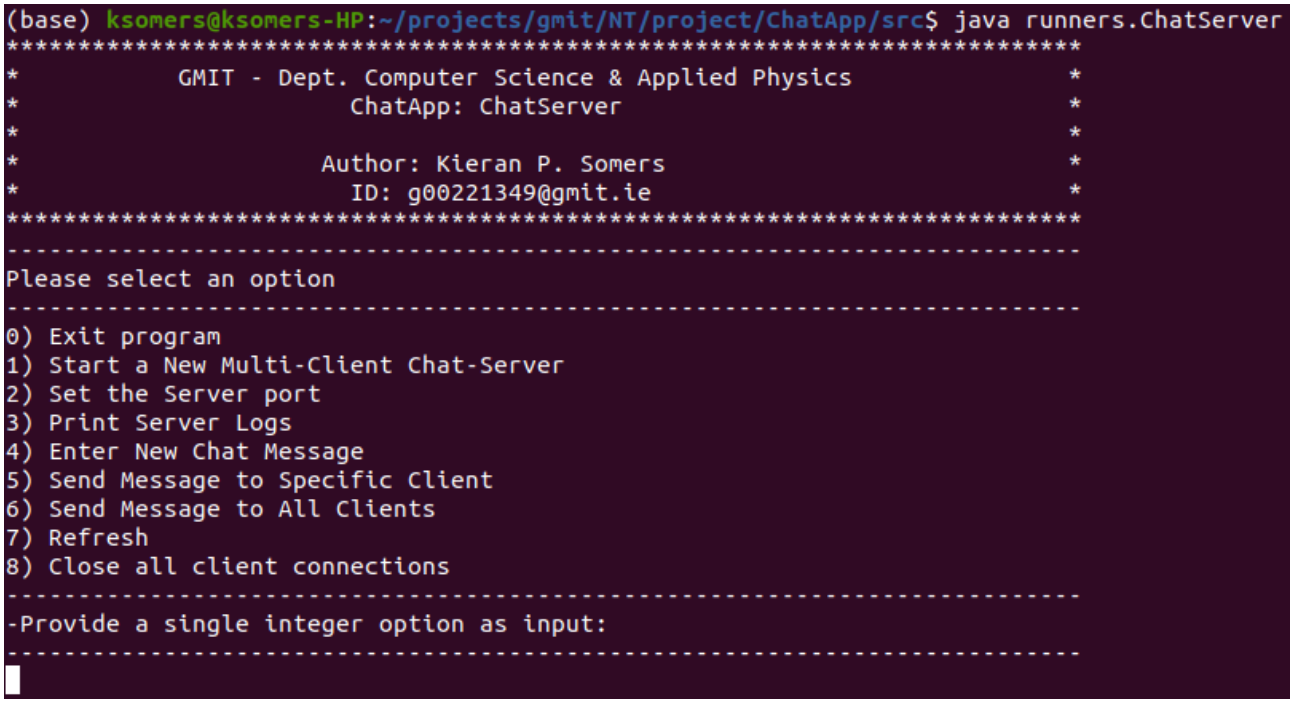

Figure 1: Screenshot of the ChatServer options menu after launching the ChatServer app via "java runners.ChatServer".

- By default the *ChatServer* app is arbitrarily configured to start a server socket listening on port 16000. If you wish to change the port, choose option 2, type in your desired port number, and press enter.
- To start a new server, choose option 1, the default port will be used unless the user has entered a specific port number.
- Multiple servers can be started by repeatedly choosing option 1 – the port number for subsequent servers is automatically incremented from port 16000, or the previous user-entered port number, but can also be set using option 2.
- ***Note that multiple clients can connect to a single server or different servers, and clients connected to different servers are able to communicate with each other.***
- Figure 2 below gives a snapshot of the server logs which can are printed by choosing option 3 from the menu. Figure 2 shows server logs after a server has been started, but before any clients have connected. Figure 4 below shows a more detailed server log after a client has connected to the server.

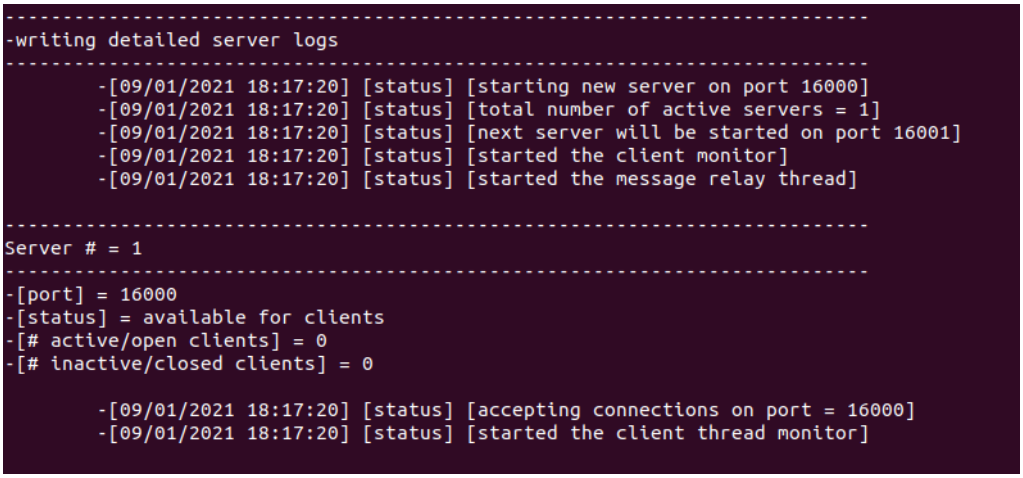

Figure 2: Screenshot of the server log after starting a server socket on 16000. The format for the events in the log is [event date time] [event type] [event message/text]. Note that no clients are connected to server # 1. Server logs can be shown be selecting option 3 from the menu.

- To send a message to one or more clients there are a number of options:
  - to enter a new chat message to send press option 4, type in your text, and press enter.
  - your input will be stored and can be sent multiple times without re-entering the message.
  - if you do not enter a message via option 4, a default message will be sent.
  - the message will not be sent until you choose option 5 or 6.
  - choosing option 5 will allow you to choose the clients to which you want to distribute the message. You will be presented with a list of clients which are currently connected to a server and with which the server can communicate. Enter a comma/hyphen separated string to choose your clients e.g. entering 1,2,3-5,7-9,15 would send the message to server/clients 1,2,3,4,5,7,8,9 and 15.
  - choosing option 6 will automatically send your message to all currently connected clients.
- After any option is chosen from the menu (with the exception of printing the server logs, option 3), a list of received messages are printed to the screen. Alternatively, one can choose option 7 (refresh) from the menu to see any received messages.

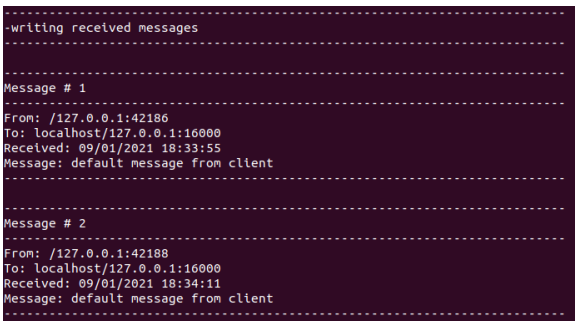

Figure 3: Screenshot of the server terminal displaying received messages after running any option other than option (other than option 3) or after option 7 (refresh) has been chosen. In this case the server has received two messages from two different clients (note the different "From" addresses) which are connected to the same server (note the same "To" addresses).

- To stop connections to clients choose option 8 from the option menu or choose option 4 to enter a new chat message and enter "\q" as your message.
- Choosing option 0 will close the entire server application, including safely closing all client connection.

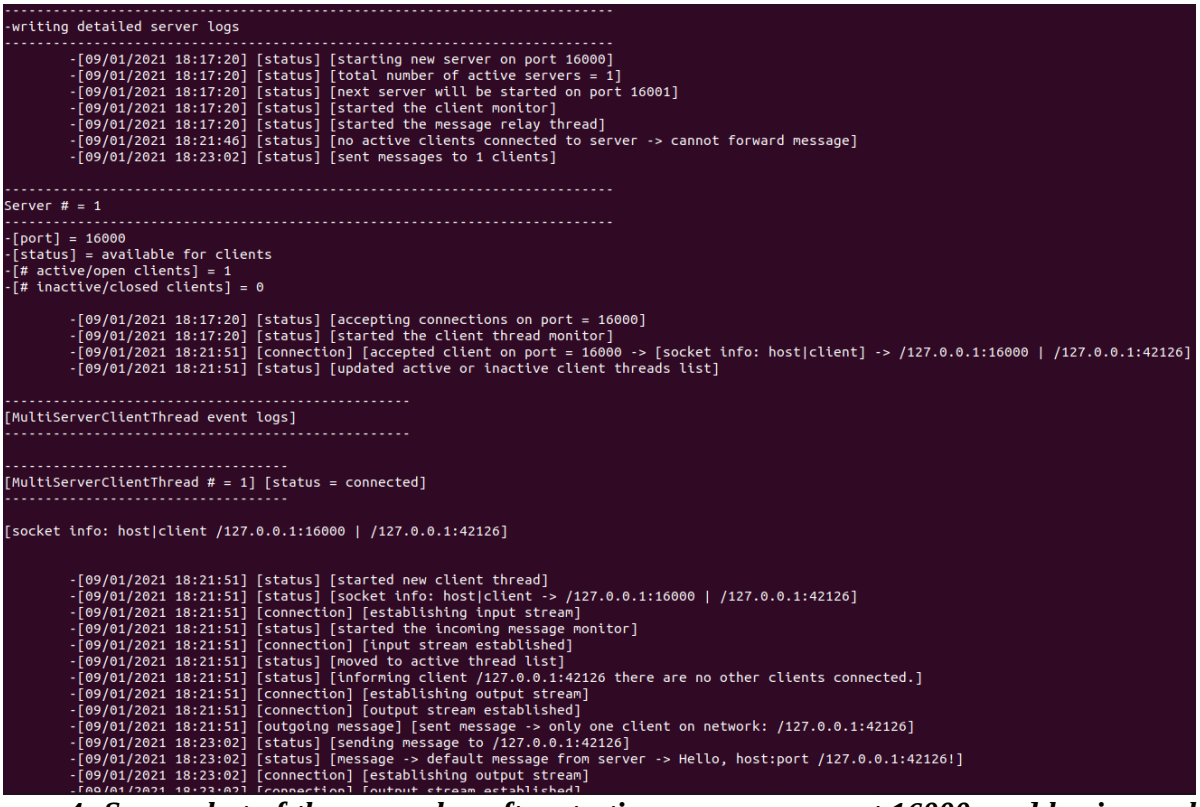

Figure 4: Screenshot of the server log after starting a server on port 16000, and having a client connect to the server. Server logs can be shown by choosing option 3 from the ChatServer menu.

ChatClient (Multiple Instances can be Launched)

Open a terminal in the *src/* or *bin/* directory (depending on whether you have compiled the source code yourself or using an IDE like eclipse) and launch the application via:

- *java runners.ChatClient*

You will be presented with an option menu as follows:

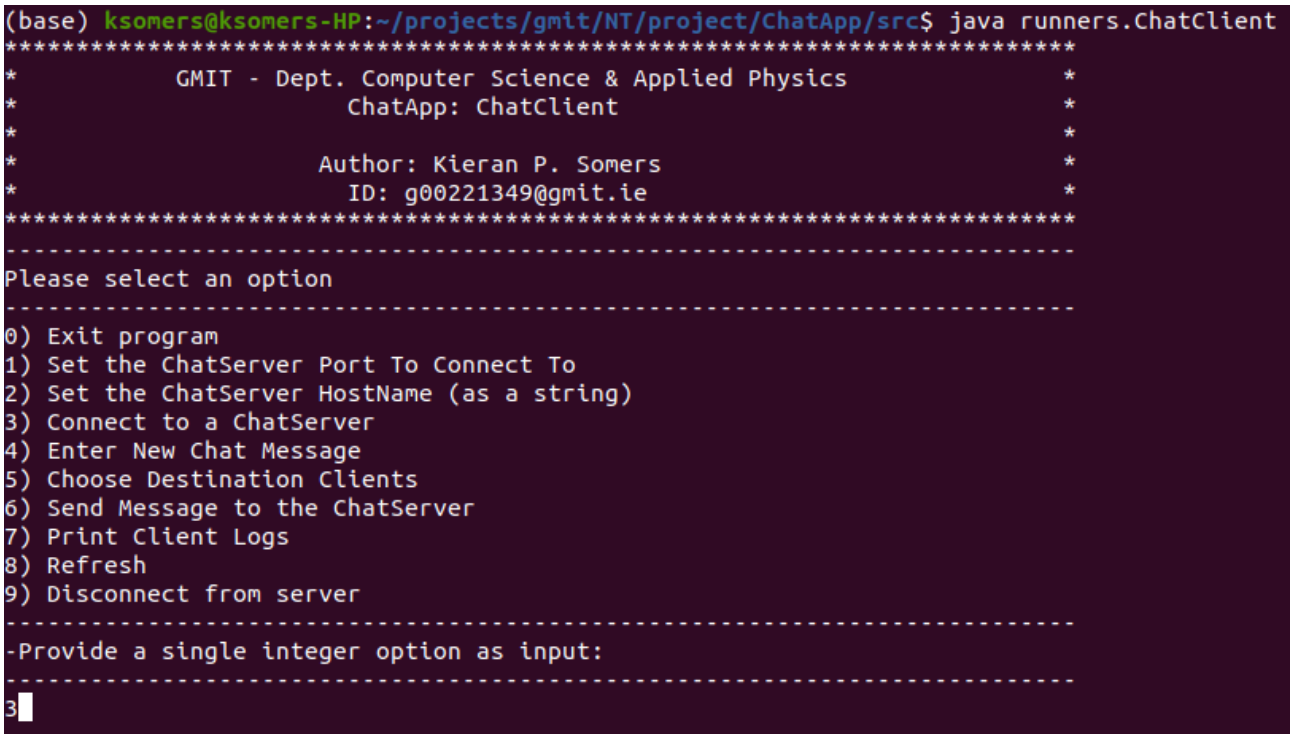

Figure 5: Screenshot of the ChatServer options menu after launching the ChatServer app via "java runners.ChatClient".

- By default the ChatClient app is configured to connect to 'localhost:16000',
- If you have started your ChatServer on a different port then you can change the port to which you will connect by selecting option 1, you will be then asked to enter the port to which you want to connect – type in the port number and press enter.
- Similarly you can change the host to which you want to connect by pressing option 2 and typing in your hostname (as a string) and pressing enter. The *ChatClient* was tested using the DayTime server (hostname:port of test.rebex.net:13) and it functioned correctly.
- Note that option 1 and 2 have no effect if the client is already connected to a server.
- To connect to a ChatServer (or more generally the host that was entered) choose option 3.
- To send a message to the server or another client there are 3 steps:
  - to enter a new chat message press option 4, type in your text, and press enter.
  - your input will be stored and can be sent multiple times without re-entering the message.
  - the message will not be sent until you choose option 6.
  - if you do not enter a message via option 4, a default message will be sent.

- If you wish to change the server/clients to which the current message will be sent choose option 5. You will be presented with a list of servers/clients which are currently connected to the server and to which you can send your message. Enter a comma/hyphen separated string to choose your clients e.g. entering 1,2,3-5,7-9,15 would send the message to server/clients 1,2,3,4,5,7,8,9 and 15.
- select option 6 to send the message(s) to the server/client(s).

- To disconnect from the *ChatClient* choose option 9 from the option menu or choose option 4 to enter a new chat message and enter "\q" as your message.
- Any received messages will be displayed after any option is chosen from the menu (with the exception of option 7 which prints client logs), or you can choose option 8 to refresh the screen and view your received messages – see figure 6 below.
- Choosing option 7 ("Print Client Logs") will print a detailed set of information about events that have occurred including connection status, other status updates, incoming and outgoing messages etc – it is useful for monitoring events and errors – see figure 7 below.
- Choosing option 0 will exit the *ChatClient*, after safely closing the connection to the server.

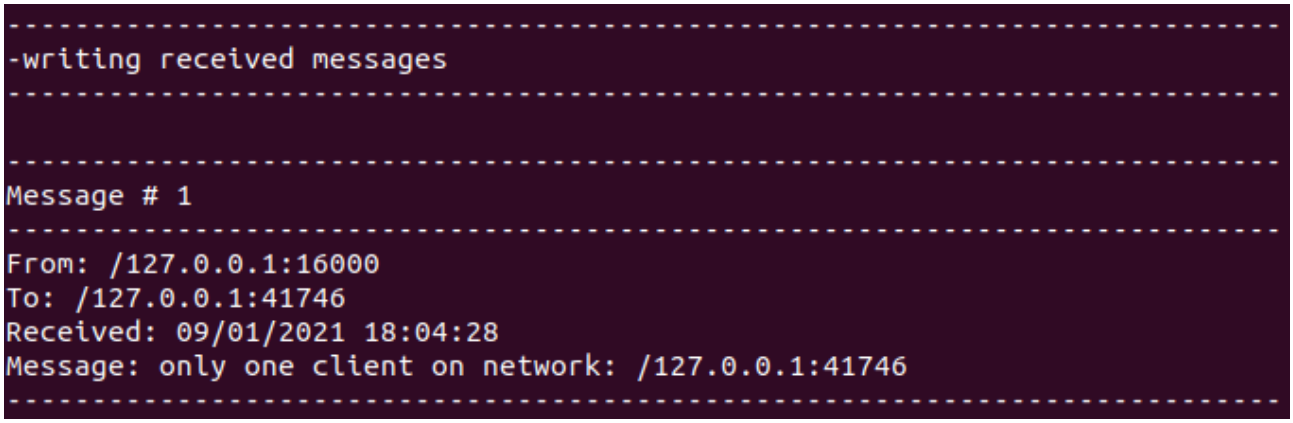

Figure 6: Screenshot of the ChatClient terminal after pressing option 8 (refresh) – a list of received messages are printed after any option is chosen (apart from option 7 which prints client logs).

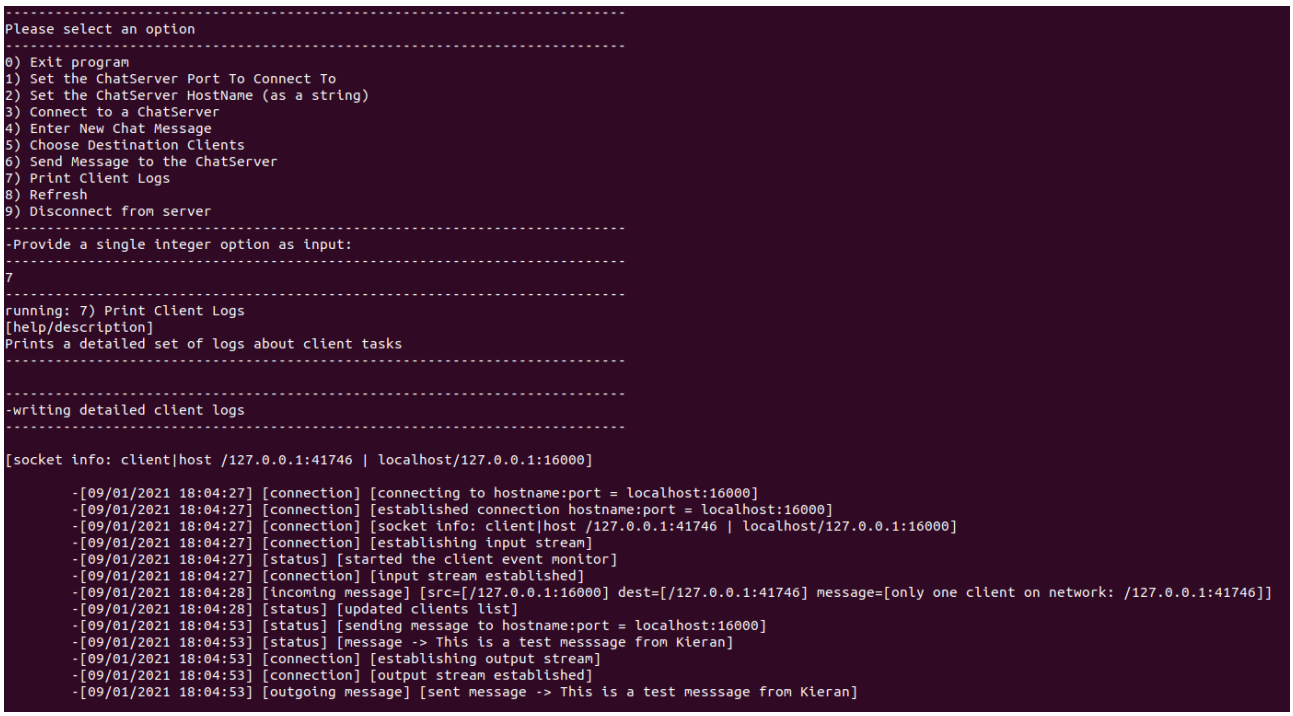

Figure 7: Screenshot of the client log after connecting to localhost on port 16000. An incoming message was received from the server after connecting to it at 18:04:28 and an outgoing message was sent from the client to the server at 18:04:53.

References

The basic reference material for the client-server architecture were the lecture notes provided by the Lecturers Sean Duignan/John French and this forms a basis for the *Client*, *MultiClientServer, StreamReader and StreamWriter* classes, with some modification by this author. The course notes were supplemented with material from Oracle Java Tutorials (see references [1-5] below) for implementing a simple client/server application, and a more complex multi-client server with server-threads (i.e. the *MultiClientServerThread* class which the author designed*)*. Some basic regular expressions/pattern matching were implemented as part of message parsing routines and source code was adapted from reference [6] below. StackOverflow [7] was consulted for basic debugging issues related to threading/concurrency, although no source code was explicitly taken from those boards.

[1]https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html
[2]https://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockClient.java
[3]https://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockServer.java
[4]https://docs.oracle.com/javase/tutorial/networking/sockets/examples/KKMultiServer.java
[5]https://docs.oracle.com/javase/tutorial/networking/sockets/examples/KKMultiServerThread.java
[6]https://www.tutorialspoint.com/java/java_regular_expressions.htm
[7]https://stackoverflow.com/