CS2003 Week 11 Practical: Web Messenger

Student ID: 150012297

November 24, 2015

Introduction

The aim of this practical was to build a peer-to-peer messaging application. This application is split in two halves: a user interface, and a message agent. The user interface communicates with the message agent using a websocket, and message agents communicate with each other over a TCP connection.

My Design

For my submission I implemented all seven of the basic requirements, and the following four extensions:

1. Evidence of my system interoperating with multiple other systems. (Extension I)
2. Persistence such that messages that have been sent previously are saved by the MA, even when the UI is not running. (Extension VI)
3. Additional persistence such that all messages are saved at the MA side when the MA is running, and if it has to be restarted, it can see the previous messages it has received also. (Extension VII)
4. An addition to the communication packet protocol which uses a special "connection_failed" packet to indicate if a user could not be connected to. (Extension VIII)

The following is a basic outline of how my implementation works, and justifications for my design decisions:

When the message agent is run it does two main things. Firstly, it sets up a message agent server to connect to other message agents over a TCP connection. Secondly it sets up an HTTP server and runs it on a specified port. When clients connect to this server it sends them a dynamically generated HTML file using Javascript, which then calls a few other files (`message.css`, `message-client.js`) which are then sent to the client. The reason for dynamically generating this file is that it gives us more flexibility regarding its contents. For instance, the page contains a list of class users which is read from a file. Should that file change, our UI will now change too.

`message-client.js` contains all the client-side processing that is done, and that includes setting up the websocket connection with the message agent. This allows for full duplex communication between the two, which is ideal for passing messages backwards and forwards. The file contains all the logic for interacting with the UI, as well as what happens when its half of the websocket receives packets (my packets match the JSON format in the specification) and sends packets. When the user presses the send button the information from the UI is taken and turned into the correct packet format. The opposite process takes place when a packet is received.

As we only wish to add sent messages to the message board if we receive a "delivery" packet, all sent messages are cached (using their unique ID as a key) and if successful they are added to the board, and if not they are removed from the cache. My fifth extension allows the user interface to identify if a recipient's message agent is offline and respond accordingly. For this I use the following packet format:

`{ type: connection_failure, payload: { receiver: <string>, id: <integer> }}`

The purpose of the "receiver" payload field is so that the UI can show the user the username which they failed to connect to, and "id" is used to remove the failed message from the cache. Figure 1 shows the UI in action and also what happens when a connection to a user fails.

When the message agent receives a packet from the user it checks the packet type to make sure it is correct and then calls a function to handle the packet data (`handleUIPacket()`). This function has the job of dealing with connecting to other (host) message agents. It looks at the recipient in the packet and sees if it currently has an open TCP connection with them. If so it sends the packet across that connection. If not, it sets one up and keeps track of that connection.

The way in which my program keeps track of these connections is one of the interesting features of my design. Whereas some of my classmates only kept one TCP connection active at once, and simply destroyed it an rebuilt it every time a message was exchanged, I decided that because the user would often be exchanging messages with the same person consecutively, it made sense to do that all over one TCP connection. However users may wish to communicate with more that one other user at once, so my program needed a way to keep track of multiple TCP sockets. I did this using my `G_maHost` object, which stores multiple sockets using the connection's remote IP as a key and containing its port number, its hostname and its socket object. When the UI is closed these sockets are also closed. This is a much better design that the previously mentioned alternative, as it greatly reduces the strain on the network induced by setting up multiple unnecessary TCP connections, and hence scales better and may allow messages to travel faster.

The process of actually sending messages to other message agents is very simple once a connection has been established. All that's needed is to add the sender to the received packet and change the packet type. Again at the message agent we choose to cache the message. Upon sending the message we also set up a timeout function which is called after 3 seconds and examines the cache to determine if that message is still there. If a delivery confirmation was received then the message would have been removed from the cache and hence our timeout function would not find it. If the function does find the message still waiting in the cache, it removes it and sends an error message back to the user.

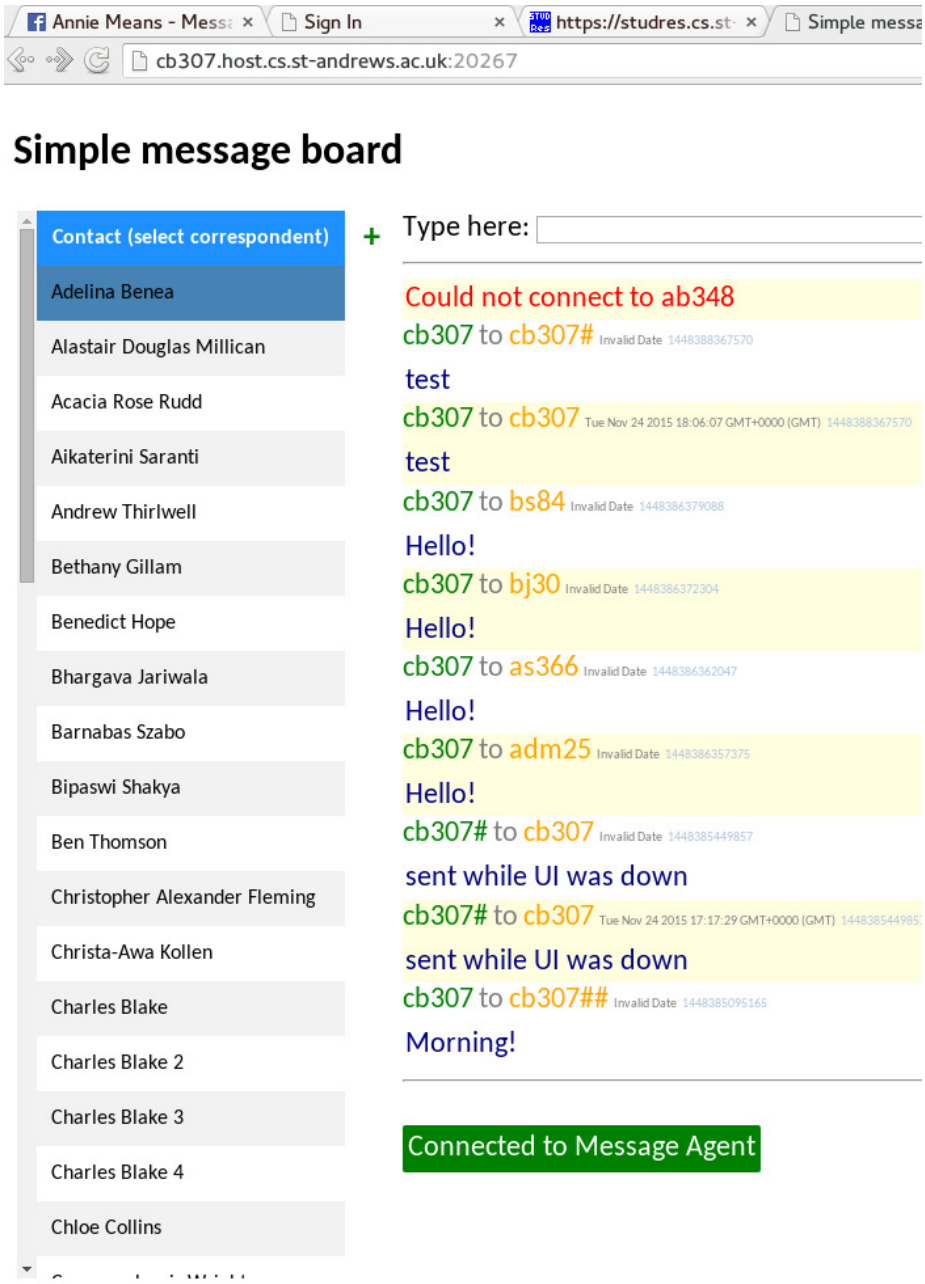

Figure 1: The user interface in action

When we receive a message a similar process happens in terms of modifying the JSON object slightly and forwarding it on to the UI, sending a confirmation back to the host MA. Another process also takes place upon receiving both messages and delivery confirmation, and that is my persistent storage feature. This extension allows users to store their entire conversation history in a file, which the message agent handles. Incoming messages are simply added to the file, and upon delivery confirmation of sent messages, the message contents are retrieved from the cache and added to the file (`conversation_history.txt`).

In addition, when the user interface initially connects to the message agent, it sends the contents of the last 10 messages to the UI. As `conversation_history.txt` stores the JSON strings which were originally sent to the UI, these 10 messages appear exactly like normal messages.

Testing

Testing this system initially seemed very difficult as the specification only talks about communicating with other classmates, and hence debugging my code by myself was going to be a difficult task. To counter this, I created several "shadow" users, who were able to operate as though they were classmates, but were actually run from my webspace (cb307), just on different ports. I did this by adding several new users to "`webs-users.txt`", with names like "cb307#" and "cb307##". These modified names were needed as my program often uses hostnames as unique identifiers, but when I actually try to connect to "cb307##.host.cs.ac.uk" I simply use a regular expression to remove the "#" symbols.

Figures 2 and 3 shows me running three message agents all on the same machine, all communicating with each other.

For a demonstration of how failed connections are treated see figure 1.

I also decided to test my implementation with other classmates to show that I had implemented the specification protocol properly. Figures 4 and 5 show my screen and imb3's screen during a web chat.

Finally I tested my persistence features to show that message are stored in a local file even when the user is offline, and are loaded when the UI opens. This is demonstrated in figures 6 and 7.

Improvements

Most of the improvements I could have made to my implementation are to do with the way it handles errors and unexpected behavior. Although my UI performs very well when receiving and sending packets which match the specification format, I haven't tested how they respond when unusual packets are received which contain incorrect or missing fields. My application can deal with MAs and UIs disconnecting, but I have not tested exactly what happens if the UI or host MA disconnect while sending packets or perhaps waiting on

confirmations. These deficiencies are not large though, and exist primarily because I chose to prioritise implementing extra functionality over very tight error handling, which was probably a reasonable use of my time. I could also have improved the CSS and general design for my UI if I wanted the program to look more professional.

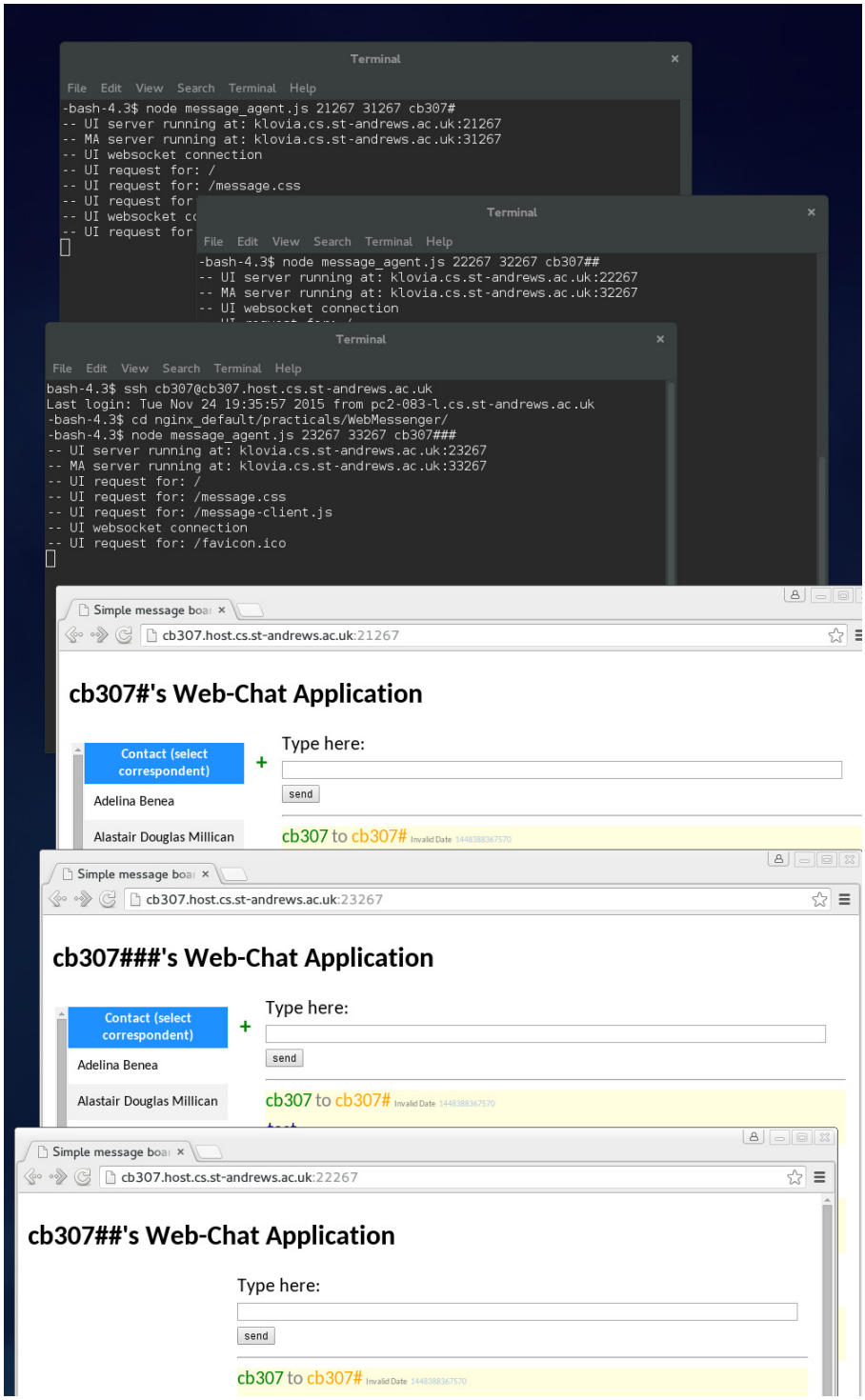

Figure 2: Setting up three message agents. Note how the usernames and port numbers shown in the 3 terminals correspond to those in the three browsers

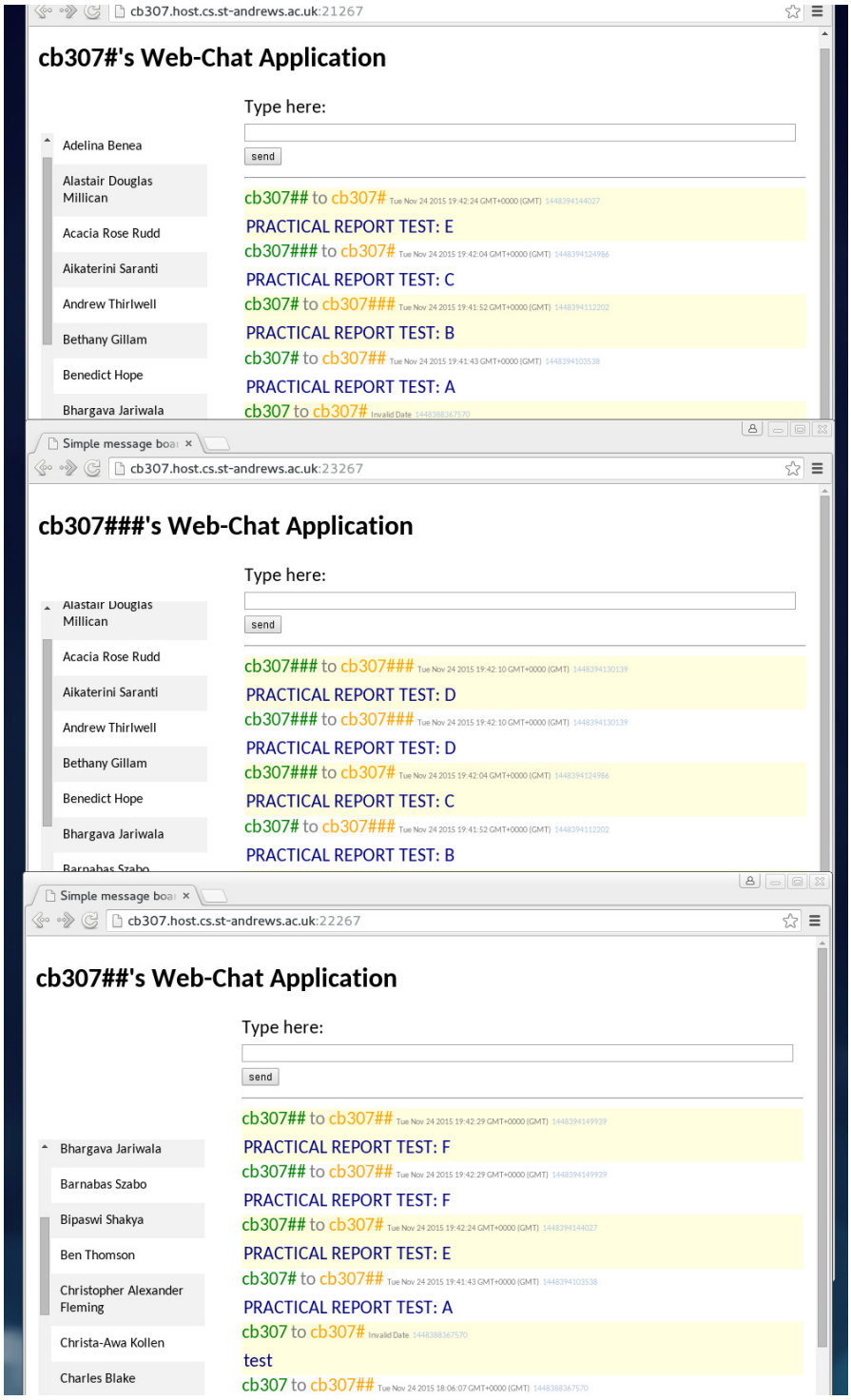

Figure 3: Three-way intercommunication between applications. Note the senders and recipients in each message table. The message agents connect using three TCP streams which are maintained throughout.

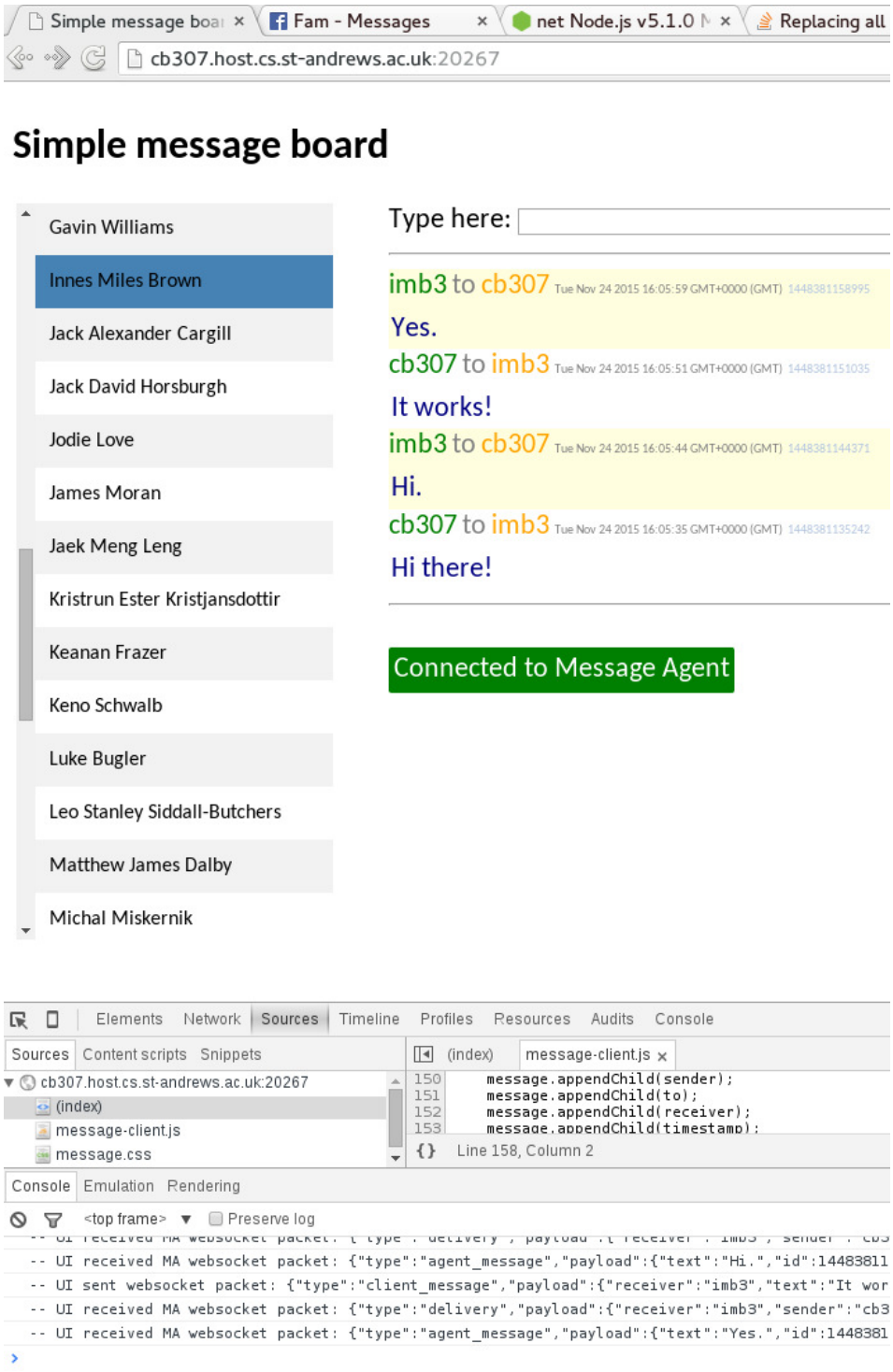

Figure 4: cb307 communicating with imb3

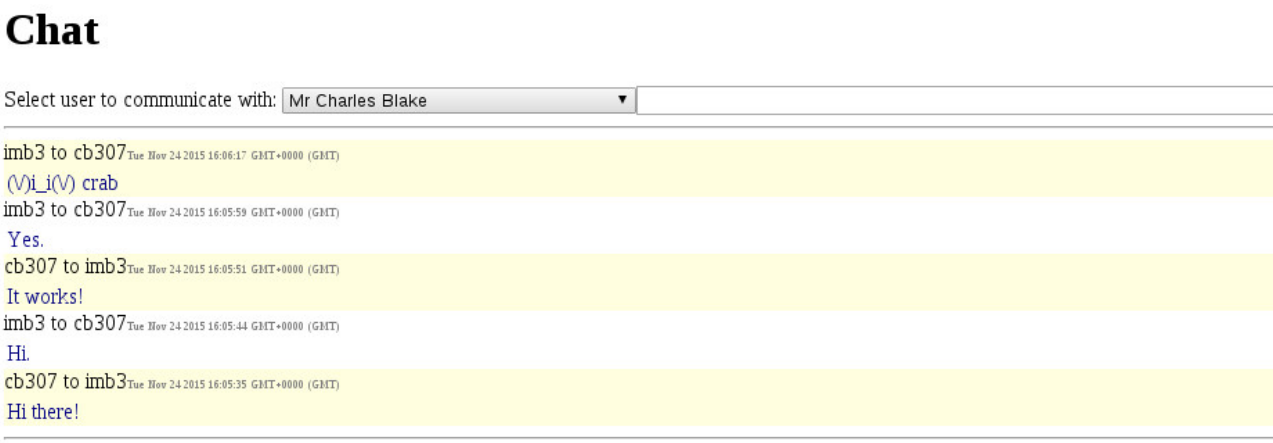

Figure 5: imb3 communicating with cb307

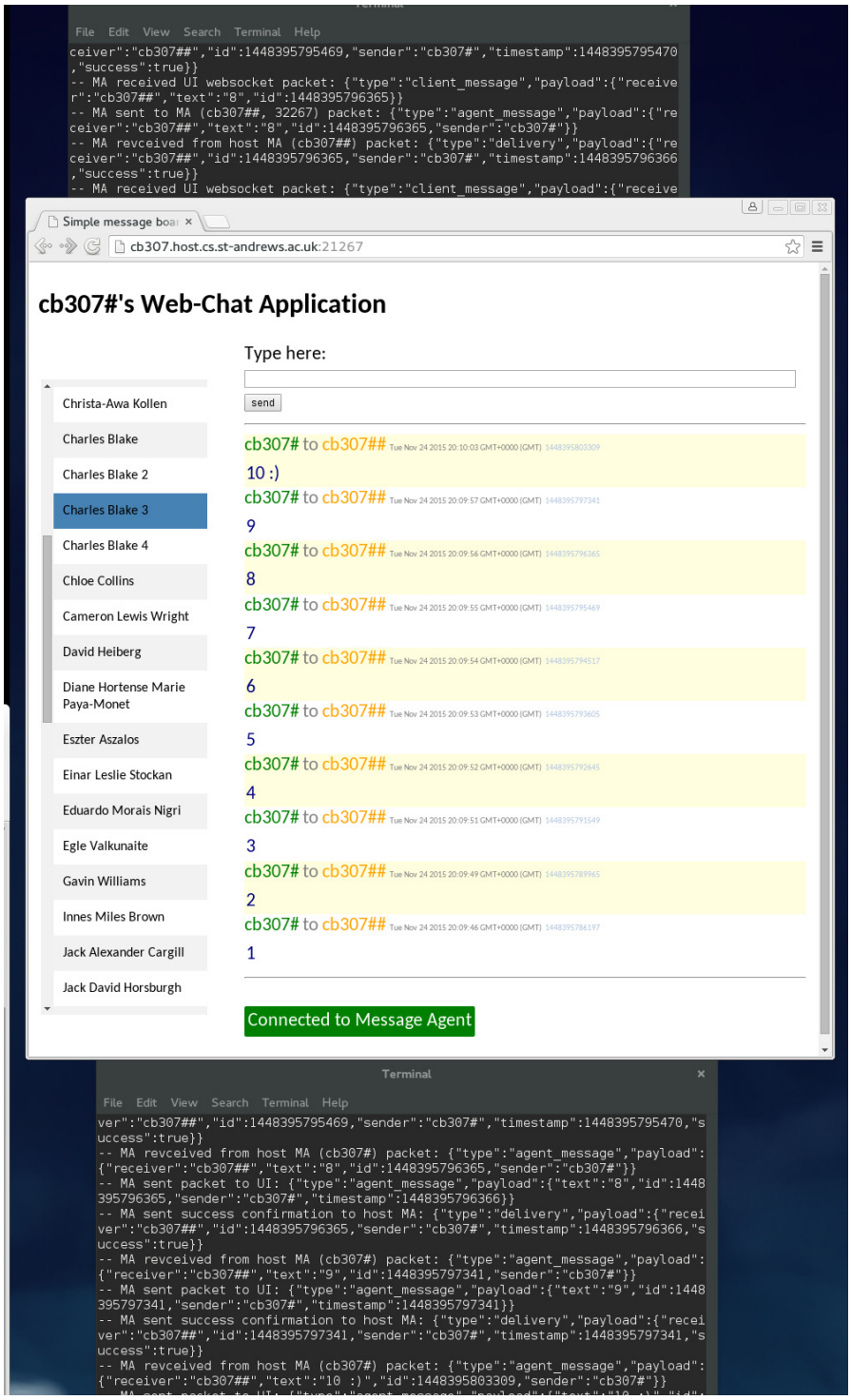

Figure 6: This shows cb307# attempting to communicate with cb307##. As we can see, there are two MAs running in the terminals, but cb307## does not have a UI running. Figure 7 shows what happens when cb307# opens his UI

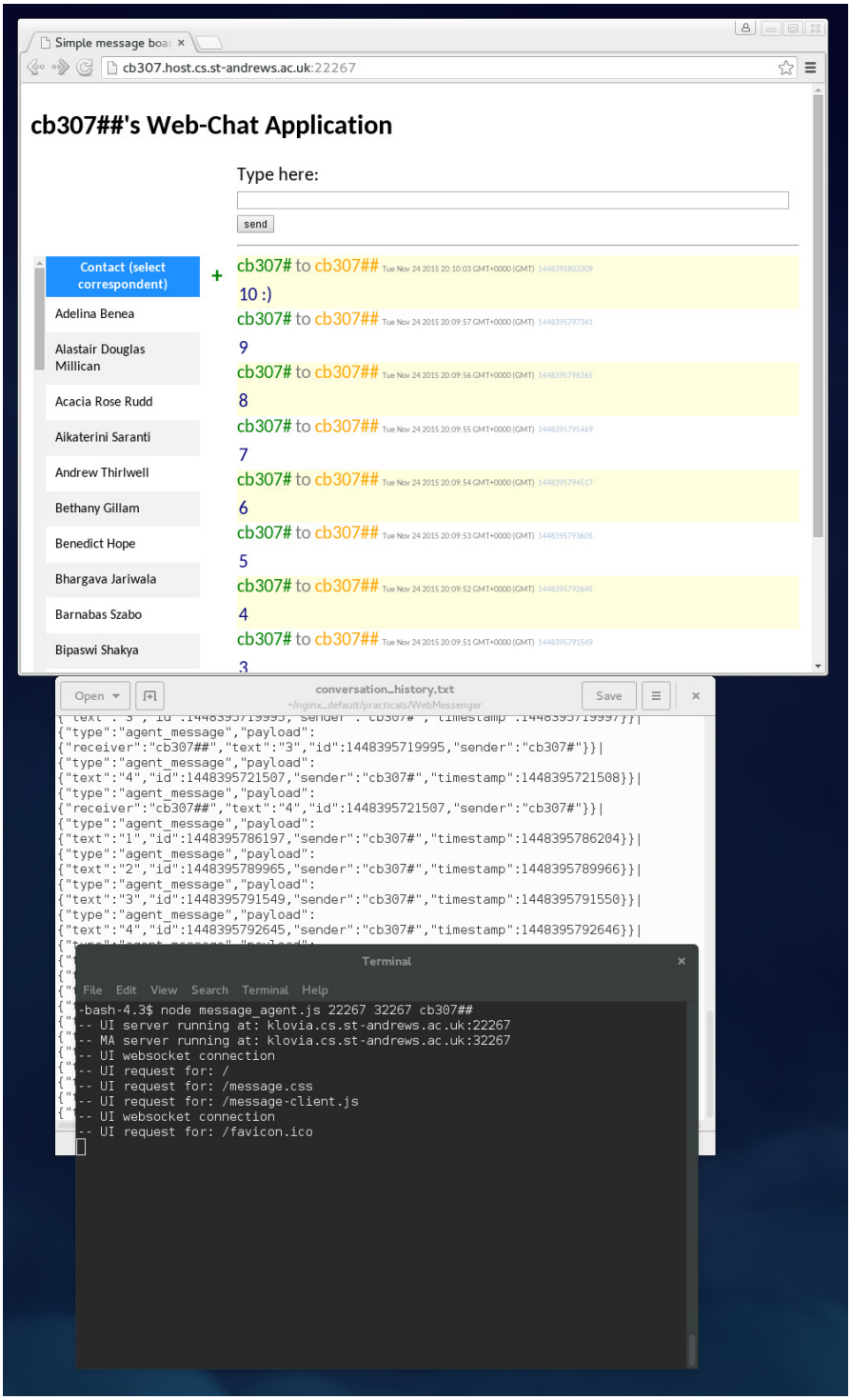

Figure 7: We can see from the terminal log that cb307## has not sent or received any messages from other users, yet the MA was able to populate his message board with the last 10 messages he recieved. Moreover, these were messages he recieved while his UI was offline (see figure 6). We can also see here the text file in which the conversation history is stored.