## Programming Assignment 1 - UDP Chat

David Jacob – Erasmus Student Awaiting ID

Because messages are not encrypted or signed we do not have confidentiality or integrity of the messages.

The client server application supports five types of requests, /join, /list, /tell, /broadcast, /leave. The join request is processed when the client starts up.

The invocation style I have chosen is “at-least-once”.

## At-least-once innvocation style

The server sends an Acknowledgement Message (ACK) for requests. The client resends requests until an acknowledgement is received. Note that messages could be delivered twice if the acknowledgement is lost.

The method used to implement the “at-least-once” invocation style is called sendChatMessageAtLeastOnce() and is in the ServerConnection class.

// implements the "at-least-once" invocation style

**public** **void** sendChatMessageAtLeastOnce(String message) {

ackReceived = **false**;

**while** (!ackReceived) {

System.***out***.println("Sending message");

sendChatMessage(message);

// leave time for an ACK

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

After sending a message, the client waits for 1 second to give time for the acknowledgement to come back. The server sends an ACK back as follows.

// send Acknowledgement

String sender = getNameByAddressPort(inetAddress, port);

server.sendPrivateMessage("ACK", sender);

## handshake()

Because of at-least-once innvocation style, it is necessary to deal with the case when /join might be called twice. The handshake method repeatedly sends the /join request and the SocketListener class deals with the case when a /join might be sent twice. This is done using a SocketTimeOut Exception. No Acknowledgements are used here.

**public** **boolean** handshake(String name) {

// retransmit, wait for 2 seconds before resending request

**try** {

socket.setSoTimeout(2000);

} **catch** (SocketException e1) {

e1.printStackTrace();

}

**while** (**true**) {

sendChatMessage(message);

**try** {

socket.receive(receivePacket);

**break**;

} **catch** (SocketTimeoutException e) {

System.***out***.println("Socket timeout");

// loop to top of while and retransmit

**continue**;

} .....

In the method listenForClientMessages() in the SocketListener class on the Server side

String reply = **null**;

**if** (server.addClient(clientName, inet, port)) {

System.***out***.println("Client added");

reply = "Success : Client added";

server.broadcastEveryoneElse(clientName +

" has joined the chat room", clientName);

} **else** {

// Note - Client could already exists because of lost reply

// check if same or different inet and port

System.***out***.println("Client already exists");

ClientConnection cc = server.getClientConnectionByName(clientName);

**if** ((inet.equals(cc.getAddress())) && (port == cc.getPort())) {

// reply got lost

System.***out***.println("Resending success message");

reply = "Success : Client added";

} **else** {

// name really is taken

System.***out***.println("Sending Error message");

reply = "Error, client already exists";

}

### Reflections

I found this a very interesting but difficult assignment. I now have a better understanding of network programming in Java and how UDP operates. I realise it is difficult to build reliability into a UDP application where it is important that messages are not lost.

As an Erasmus student I was accepted into the course late through registration and I found it difficult to get it done within one week.