CSC4010 Assignment 2

Cameron McGreevy

Overall Design

The system provides a peer to peer network that users can use to chat on and send text files that has been created using Java. When a node joins the network it automatically sends out a broadcast packet and is discovered by all other nodes on the network. The clients respond with a packet of their own with details of their node and adds the node to a list of registered nodes on the network, as does the incoming user. Now all nodes are connected and aware of each other, messages sent out from any node are sent to all nodes and processed against an input list. If the message is a regular text message it will be compared against each entry in the existing chat history and slotted into place based on the timestamp of the creation of the message. Additional operations are also available that I shall detail further on in the report and include the ability to send text files to other users on the network, create, view and join other chat rooms on the network that are isolated from each other but discoverable for each node and the ability to synchronise chat messages.

Node Discovery, Connection and Chat History

When joining the network, users only have to launch the program with no need to put in their local address or port as they will automatically be assigned one and then discovered by other nodes using the 'Broadcast' function of UDP. When the node starts up, the Listener thread that receives incoming traffic is the first to start and a broadcast packet is sent from this thread using the socket it listens on to '255.255.255.255' whereby default all nodes will be listening.

```
// Send out an initial packet to broadcast address for existing nodes to find
chatrooms.put(this.node.getcRoom(), 1);
ds.setBroadcast(true);
// 255.255.255.255 is the designated broadcast address
InetAddress bAdd = InetAddress.getByName("255.255.255.255");
// Send out information
String tStamp = "/INFO@" + this.node.getNickname() + "@" + this.node.getTime() + "@" + this.node.getcRoom();
System.out.println(tStamp);
DatagramPacket dp = new DatagramPacket(tStamp.getBytes(), tStamp.length(), bAdd, 5000);
ds.send(dp);
// create a byte array of size 65535 to receive data
byte[] receive = new byte[65535];

// packet for receiving data
DatagramPacket packet = null;
```

This is followed by the Broadcast thread which listens on port 5000 for incoming broadcasts and finally the Sender thread which handles sending messages.

```
// Called in StartNode to start the node
public void Start(Node localNode) throws UnknownHostException, SocketException {
    System.out.println(localNode.getNickname() + " has connected");
    // Adds itself to the room and client list
    clients.add(localNode);
    room.add(localNode);
    Broadcast broadcaster = new Broadcast(localNode);
    Sender send = new Sender(localNode);
    Listener listen = new Listener(localNode);
    // Listener is started first so that it can send the broadcast packet which is then picked up by any existing nodes on the network
    listen.start();
    // Broadcaster thread starts to listen to any new nodes
    broadcaster.start();
    // Finally start thread to send messages
    send.start();
}
```

The sent broadcast packet will contain details about the new node such as its nickname and the timestamp of its creation. Existing nodes receiving this will then parse the packet and create a new node to be stored in its connection list with the exact properties of the incoming node with its IP address and socket details parsed from the packet metadata.

```
// A thread that listens on port 5000 for broadcasts from nodes wanting to join the network
public class Broadcast extends Thread {
    private byte[] buf = new byte[1024];
    Node node;
    boolean newNode = true;

    public Broadcast (Node lNode) {
        this.node = lNode;
    }

    public void run() {
        try {
            DatagramSocket socket = new DatagramSocket(5000);
            while (!this.node.quit) {
                System.out.println("Server waiting for broadcast");
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                socket.receive(packet);
                String data = ByteToString(buf);
                // Split incoming message with '@' symbol and parse
                String[] details = data.split("@");
                // Nickname of node wanting to join
                String newNN = details[1];
                // Timestamp of new node
                long timeS = Long.parseLong(details[2]);
                // Chatroom of node wanting to join
                String nodeChatRoom = details[3];
                if (newNN.equals(this.node.getNickname())) {
                    // Skip if trying to add itself
                    continue;
```

If the incoming node is in the same chat room as the existing one it will be stored in both the chatroom list and the list containing all clients on the network. After this it replies with a packet of its own containing its own IP address and port (these cannot be parsed from the packet metadata as it is being sent from the Broadcast socket), alongside it's nickname, timestamp and chat room. The new node receives this packet from each client already on the network and repeats the same process for adding new nodes to its stored lists.

```
for (Node n: clients) {
    // If already in node list, skip
    if (n.getcNick().equals(newNN) && n.getTime() == timeS && n.getcIp() == packet.getAddress() && n.getcPort() == packet.getPort())
        newNode = false;
}
if (newNode) {
    for (Node n: clients) {
        // Skip if adding itself
        if (n.getcIp() == this.node.getcIp() && n.getcPort() == this.node.getcPort() && n.getTime() == this.node.getTime()) {
            continue;
        } else {
            // Send details of new node onto other nodes with /ADD tag
            String aPkt = "/ADD@" + packet.getAddress() + "@" + packet.getPort() + "@" + newNN + "@" + timeS + "@" + nodeChatRoom;
            DatagramPacket dp = new DatagramPacket(aPkt.getBytes(), aPkt.length(), n.getcIp(), n.getcPort());
            ds.send(dp);
        }
    }
    Node newClient = new Node(packet.getAddress(), newNN, nodeChatRoom);
    newClient.setcPort(packet.getPort());
    newClient.setTime(timeS);
    // Add to overall client list
    clients.add(newClient);
    // Send the new node this node's details
    String dPkt = "/INFO@" + this.node.getNickname() + "@" + this.node.getTime() + "@" + this.node.getcRoom();
    DatagramPacket dp = new DatagramPacket(dPkt.getBytes(), dPkt.length(), packet.getAddress(), packet.getPort());
    ds.send(dp);
    // Check if the new node is in the chat room
    if (this.node.getcRoom().equals(nodeChatRoom)) {
        System.out.println(this.node.getNickname() + " has joined the room");
        // Add to room list
        System.out.println("Before broadcast add: " + room.size());
        room.add(newClient);
        System.out.println("After broadcast add: " + room.size());
        // Increase population of chatroom
        int currVal = chatrooms.get(nodeChatRoom);
        chatrooms.put(nodeChatRoom, currVal + 1);
```

Finally, if the existing nodes are in the same chatroom as the incoming node, they send a copy of their chat history to the node joining the network. Each Message object stored in a nodes chat history list is broken up with dividers and added to a string which is then sent in a single packet to the new node which uses the java String.split() function to first parse the string using special characters for individual messages and then parses it again for the components of each message. A new Message object is created and stored in the chat history list. When multiple nodes send their chat histories, the node takes the first received list and compares each new list to check for missing messages and adds them in at the correct spot if detected.

```
// Check if the new node is in the chat room
if (this.node.getcRoom().equals(nodeChatRoom)) {
    System.out.println(this.node.getNickname() + " has joined the room");
    // Add to room list
    System.out.println("Before broadcast add: " + room.size());
    room.add(newClient);
    System.out.println("After broadcast add: " + room.size());
    // Increase population of chatroom
    int currVal = chatrooms.get(nodeChatRoom);
    chatrooms.put(nodeChatRoom, currVal + 1);
    // Send all messages in history as one string with characters to help parse later
    String msgsToSync = "";
    for (Message m: messages) {
        msgsToSync += m.getmSender() + ">" + m.getmMessage() + ">" + m.getmTStamp() + ">" +m.getMessageID() + "~";
    }
    // Send message history to new node
    String msgPkt = "/MSG@" + msgsToSync;
    System.out.println("Synced: " + msgPkt);
    DatagramPacket msgDp = new DatagramPacket(msgPkt.getBytes(), msgPkt.length(), packet.getAddress(), packet.getPort());
    ds.send(msgDp);
} else if (chatrooms.containsKey(nodeChatRoom)) {
    // If not in the same room, just increase value of room population
    int currentVal = chatrooms.get(nodeChatRoom);
    chatrooms.put(nodeChatRoom, currentVal + 1);
} else {
    // If a new chatroom, add it to hash map
    chatrooms.put(nodeChatRoom, 1);
}
```

By default, nodes created with no arguments will join the ‘default’ chatroom, but nodes can also be created with an argument which will allow them to join or create a chatroom by that name when they start up. The same process as above is followed but it will check if a chat room by the name it goes by exists before sending packets.

Parsing Messages

Messages are received on the Listener thread and are sent in the format of a string with each component separated by an ‘@’ symbol. The string is then broken up when received and the first component is checked against an if-else block. This may contain a word that is used for functionality such as ‘/CREATEROOM’, and if so this is recognised and the function executed. Otherwise messages are assumed to be text messages sent to other nodes and added to the message history.

```
while (!this.node.quit) // keep going until quit
{
    System.out.println("CLIENT waiting for data");

    // create the packet to receive data
    packet = new DatagramPacket(receive, receive.length);
 // receive the data into the packet
    ds.receive(packet);
    String data = ByteToString(receive);
    String[] details = data.split("@");
   for (String sr: details) {
       if (sr.equals("localhost/127.0.0.1")) {
           sr = "127.0.0.1";
        }
   }
```

Node Properties

- Each node contains its own unique set of properties
  - A nickname, initially generated by a random number
  - The IP of the network it is on
  - A Boolean denoting if the node's chat log is synced or not
  - A datagram socket assigned to a random unused port
  - The port the socket is using
  - The unix timestamp at which it was created
  - A list of messages from the overall chat log
  - A list of local messages sent by itself
  - A hash map containing every chatroom and the number of people in it
  - Which chat room it is in
  - The number of messages it has sent

```
// Two constructors, first joins default chatroom, second joins a specified chatroom
public Node(InetAddress ip, String nickname) {
    this.cNick = nickname;
    this.cIp = ip;
    this.cSynced = false;
    try {
        this.ds = new DatagramSocket(0);
    } catch (SocketException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    this.cPort = ds.getLocalPort();
    this.quit = false;
    this.setTime(System.currentTimeMillis() / 1000L);
    this.messages = new ArrayList<Message>();
    this.localMessages = new ArrayList<Message>();
    this.chatrooms = new HashMap<String, Integer>();
    this.setcRoom("default");
    this.setMessageNo(0);
}

public Node(InetAddress ip, String nickname, String room) {
    this.cNick = nickname;
    this.cIp = ip;
    this.cSynced = false;
    try {
        this.ds = new DatagramSocket(0);
    } catch (SocketException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    this.cPort = ds.getLocalPort();
    this.setTime(System.currentTimeMillis() / 1000L);
    this.messages = new ArrayList<Message>();
    this.chatrooms = new HashMap<String, Integer>();
    this.setcRoom(room);
    this.setMessageNo(0);
}
```

- Node nicknames are initially set by random number generation, but can be changed by the user
  - The user inputs '/nickname' and then enters a new nickname

- A check is made of existing nodes to see if the name is already in use and is appended with a number at the end if it does e.g., Nickname (1)
- The new nickname is then sent out to all nodes on the network including this node in a message identifiable with the keyword '/NICKNAME'
- When received, the node changes its nickname and retroactively changes the message history to reflect this
- Other nodes on the network find the node in their client lists based on its IP, port and timestamp and do the same

```
} else if (input.equalsIgnoreCase("/nickname")) {
    System.out.println("");
    System.out.print("Enter Nickname: ");
    String input2 = sc.nextLine();
    int nicknameCount = 0;
    for (Node c: clients) {
        if (c.getNickname().equals(input2)) {
            nicknameCount += 1;
        }
    }
    if (nicknameCount > 0) {
        message = "/NICKNAME@" + input2 + "(" + nicknameCount + ")@" + this.node.getNickname() + "@"+ this.node.getTime();
        this.node.setNickname(input2 + "(" + nicknameCount + ")");
    } else {
        message = "/NICKNAME@" + input2 + "@" + this.node.getNickname() + "@" + this.node.getTime();
        this.node.setNickname(input2);
    }
// Handles requests by nodes to change their nickname
} else if (details[0].equals("/NICKNAME")) {
    // Set stored node's nickname
    for (Node n: clients) {
        if (n.getNickname().equals(details[2]) && n.getTime() == Long.parseLong(details[3])) {
            n.setNickname(details[1]);
        }
    }
    // Change the nickname for senders in message history
    for (Message m: messages) {
        if (m.getmSender().equals(details[2])) {
            m.setmSender(details[1]);
        }
    }
```

Quitting

- A node can quit by sending '/quit'
  - A message is sent to all nodes on the client list with the '/QUIT' keyword
  - When received by itself, the node will stop all while loops and stop the program
  - When received by other nodes they remove the client from their list, and room if necessary, also updating the chatroom hashmap to reflect that a member has left a room
- If any node leaves unexpectedly messages can still be sent and functionality is not affected

```
if (input.equalsIgnoreCase("/quit")) {
    System.out.println("Shutting Down");
    message = "/QUIT@" + this.node.getNickname() + "@" + this.node.getTime() + "@" + this.node.getcPort() + "@" + this.node.getcIp();
    this.node.quit=true;
```

```
// Ends client session
} else if (details[0].equals("/QUIT")) {
    for (Node quitter: clients) {
        if (details[1].equals(this.node.getNickname()) && Long.parseLong(details[2]) == this.node.getTime()) {
            System.exit(0);
        }
        if (details[1].equals(quitter.getNickname()) && quitter.getTime() == Long.parseLong(details[2]) && Integer.parseInt(details[3]) == quit
            clients.remove(quitter);
            if (room.contains(quitter)) {
                room.remove(quitter);
                System.out.println(details[1] + " has left room");
                int currentVal = chatrooms.get(this.node.getcRoom());
                chatrooms.put(this.node.getcRoom(), currentVal - 1);
            }
            System.out.println(details[1] + " has disconnected");
            break;
        }
    }
```

Sending a message

- User types in a message which is stored in a string
- The message is appended with the node's nickname, the timestamp which it was sent and the message number all separated by a delimiter
- The message is sent to each node in the room list to be parsed and storede

```
} else {
    message = node.getNickname() + "@" + input + "@" + msgTS + "@" + this.node.getMessageNo();
    localMessages.add(this.node.messageNo, new Message(input, this.node.getNickname(), msgTS, this.node.getMessageNo()));
    this.node.setMessageNo(this.node.getMessageNo() + 1);
    textMsg = true;
}

// convert the string into a byte array
buf = message.getBytes();
// Cases that determine how the message is sent and who it is sent to
if (textMsg) {
    for (Node c: room) {
        DatagramPacket dp = new DatagramPacket(buf, buf.length, c.getcIp(), c.getcPort());
        sendDs.send(dp);
    }
} else if (userCommand) {
    continue;
}  else if (failuresim) {
    for (int i = 0; i < room.size(); i++) {
        if (i == failNode) {
            continue;
        } else {
            DatagramPacket dp = new DatagramPacket(buf, buf.length, room.get(i).getcIp(), room.get(i).getcPort());
            sendDs.send(dp);
        }
    }
} else {
    for (Node c: clients) {
        DatagramPacket dp = new DatagramPacket(buf, buf.length, c.getcIp(), c.getcPort());
        sendDs.send(dp);
    }
}
```

Message Storage

- Each message's details are stored in a Message object
    - Contains the sender, the message itself, the timestamp it was sent at and the sender's message number
    - When a message is received each component is parsed individually and added to the object which can then be retrieved or changed later on
- Messages are stored in an ArrayList, as it is impossible to predict how many will be sent, as the chat history
- Messages sent by a node are stored separately in a 'LocalMessages' ArrayList

- Messages are automatically sorted by their timestamp when received and placed into the message history
    - If a message has a lower timestamp than an existing stored one, it is inserted at that position

```
public class Message{
    private String mSender, mMessage;
    private long mTStamp;
    private int messageID;
    private boolean synced;

    public Message(String sender, String message, long timestamp, int msgNum) {
        this.mMessage = message;
        this.mSender = sender;
        this.mTStamp = timestamp;
        this.messageID = msgNum;
        this.synced = false;
    }
```

```
} else {
    int mSize = messages.size();
    for (int i = 0; i < mSize; i++) {
        // If existing message has been sent out of order put it back in place
        if (Long.parseLong(details[2]) < messages.get(i).getmTStamp()) {
            messages.add(i, new Message(details[0], details[1], Long.parseLong(details[2]), Integer.parseInt(details[3])));
        } else {
            if (i == mSize - 1) {
                messages.add(new Message(details[0], details[1], Long.parseLong(details[2]), Integer.parseInt(details[3])));
            } else {
                continue;
            }
        }
    }
    if (mSize == 0) {
        messages.add(new Message(details[0], details[1], Long.parseLong(details[2]), Integer.parseInt(details[3])));
    }
    // Print out the incoming message as: username> message
    System.out.println(details[0] + "> " + details[1]);
}
```

Chat Bot

- Users can enter '/chatbot' to start a dialogue generation
- The bot sends random greetings from the node it is called on for 15 seconds then stops

```
// Start chat bot
} else if (input.equalsIgnoreCase("/robot")) {
    long t= System.currentTimeMillis();
    long end = t+15000;
    while(System.currentTimeMillis() < end) {
        String[] greetings = {"Hello", "Hi", "Yo", "..."};
        Random r = new Random();
        int low = 1;
        int high = 4;
        System.out.println(this.node.getNickname() + "> " + greetings[(r.nextInt(high-low) + low) - 1]);
        Thread.sleep(3000);
        userCommand = true;
    }
```

Malformed data handling

- I have used a number of try/catch blocks to stop errors crashing the program
- If an exception is thrown they will be handled before the program can crash

```
} catch (IOException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
    quit = true;
    System.exit(1);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Peer to Peer

Please refer to the opening two sections for more details on this, but to summarise:

- All nodes are aware of each other and can contact each other at any time
- Any node can leave or join without affecting the network
- At any time no node is more important or has more authority than another node
- Messages and files are sent by a node to every other node, and there is no middle man

Failure Simulation

- Users can enter '/failsim' to generate a failure in sending a package
- A node on the network is chosen at random to not receive the message 'Failure Simulated'
- This allows for later testing of retrieving lost data

```
// Generate a random number of a node to be designated to not receive a message to simulate a failure
} else if (input.equalsIgnoreCase("/failsim")) {
    Random r = new Random();
    boolean notNode = true;
    while (notNode) {
        int low = 1;
        int high = room.size();
        failNode = (r.nextInt(high-low) + low) - 1;
        if (!room.get(failNode).getNickname().equals(this.node.getNickname())) {
            notNode = false;
        }
    }
    failuresim = true;
    message = node.getNickname() + "@" + "Failure test" + "@" + msgTS + "@" + this.node.getMessageNo();
    localMessages.add(this.node.messageNo, new Message("Failure Test", this.node.getNickname(), msgTS, this.node.getMessageNo()));
    this.node.setMessageNo(this.node.getMessageNo() + 1);
```

Chat Rooms

- Each node is aware of each other but I have split the network into chat rooms
- Nodes can only receive messages or texts if they are in the same room
- Any node can create or join a new chat room at any time
- Nodes can also view a list of chatrooms and how many people are in them using '/viewrooms'
- Creating a room
    - Users enter a name for the room and like choosing a nickname if the room exists then the name is appended with a number to distinguish it
    - A message is sent with '/CREATROOM' so that the nodes know how to handle it and contains the nodes nickname, timestamp, name of the new room and name of the room they are leaving
    - Nodes receives its own message
        - The node clears its logs and room list
        - Updates its chat room hash map, adding the new room and decrementing the value of the room it just left

        - Adds itself to the new room list
    - Other nodes receive the message
        - Nodes update their hash maps to add the new room and decrement the value of the room it left
        - Find the node creating the room in their client lists and update the value of their cRoom variable storing the room they are in
        - If in the same room as the node that is leaving, remove it from the room list and prints a message telling clients in the room that the user has left

```
// Creates a new chatroom
} else if (input.equalsIgnoreCase("/createroom")) {
    System.out.println("");
    System.out.print("Enter Room Name: ");
    String input2 = sc.nextLine();
    int roomCount = 0;
    // If the name of the room exists add a number to the end to distinguish it
    for (Map.Entry<String, Integer> set : chatrooms.entrySet()) {
        if (set.getKey().equals(input2)) {
            roomCount++;
        }
    }
    String roomName;
    if (roomCount > 0) {
        roomName = input2 + "(" + roomCount + ")";
    } else {
        roomName = input2;
    }
    System.out.println("Disconnected from " + this.node.getcRoom());
    // Send notice to other nodes on the network
    message = "/CREATEROOM@" + this.node.getNickname() + "@" + this.node.getTime() + "@" + roomName + "@" + this.node.getcRoom();
// Handles a node creating a new room
} else if (details[0].equals("/CREATEROOM")) {
    for (Node n: clients) {
        if (details[1].equals(n.getNickname()) && n.getTime() == Long.parseLong(details[2])) {
            /** If this node is the one creating a room then clear all existing messages and nodes stored in room
             *  Put new chatroom in the hashmap and decrease the value of the one that it is leaving
             *  Add itself into the new room list
             */
            if (details[1].equals(this.node.getNickname()) && this.node.getTime() == Long.parseLong(details[2])) {
                room.clear();
                messages.clear();
                int currentVal = chatrooms.get(this.node.getcRoom());
                chatrooms.put(this.node.getcRoom(), currentVal - 1);
                chatrooms.put(details[3], 1);
                this.node.setcRoom(details[3]);
                System.out.println("Joined chatroom " + details[3]);
                room.add(this.node);
            } else {
                // Otherwise, if in the same room as the node, remove it from the list and edit the hashmap accordingly
                if (this.node.room.contains(n)) {
                    System.out.println("In room");
                    room.remove(n);
                    n.setcRoom(details[3]);
                    chatrooms.put(details[3], 1);
                    chatrooms.put(details[4], (this.node.chatrooms.get(details[4]) - 1));
                    System.out.println(details[1] + " has left the chatroom");
                // Otherwise just edit the hashmap
                } else {
                    chatrooms.put(details[3], 1);
                    int val2 = chatrooms.get(details[4]);
                    chatrooms.put(details[4], val2 - 1);
                    n.setcRoom(details[3]);
                }
            }
        }
    }
```

- Joining a room
    - User enters a room name and if it doesn't exist the program advises them of this fact
    - Otherwise, a similar message is sent to that of creating a room except with the '/JOINROOM' tag to differentiate it
    - Upon receiving, nodes cycle through their client lists to find the node that is joining a new room
        - Each client's current chat room value is checked and if it is in the room that the sender is trying to join it is added to a temporary list
        - If the node finds itself

        - Clears its room list and messages list, updates chat room has map
    - Otherwise it updates its hash map and removes the node from its list if it is in the room that the sender is leaving
- After the loop, if the node receiving the '/JOINROOM' message is the same as the node that sent it, it loops through the temporary list of nodes that are already in the room it is joining
    - It adds them to the room list and sends each of them a '/ROOM' packet with its nickname and timestamp attached
    - This packet tells each of the nodes that this node is joining their room, but does not send all it's details as they already have them stored in their client list
- When the '/ROOM' packet is received each of the existing nodes in the room adds the joining one to their room lists and sends back a '/MSG' packet with their chat histories

```
// Allows a user to join an existing room
} else if (input.equalsIgnoreCase("/joinroom")) {
    System.out.println("");
    System.out.print("Enter Room Name: ");
    String input2 = sc.nextLine();
    // If room exists send notice to other nodes in the current chatroom, otherwise advise user it doesn't exist
    if (chatrooms.containsKey(input2)) {
        System.out.println("Disconnected from " + this.node.getcRoom());
        message = "/JOINROOM@" + this.node.getNickname() + "@" + this.node.getTime() + "@" + input2 + "@" + this.node.getcRoom();
    } else {
        System.out.println("Room does not exist");
        userCommand = true;
    }
// Handles a node joining an existing room from another existing room
} else if (details[0].equals("/JOINROOM")) {
    ArrayList<Node> tempList = new ArrayList<Node>();
    for (Node n: clients) {
        // Store the list of nodes in the room the node is joining
        if (n.getcRoom().equals(details[3])) {
            tempList.add(n);
        }
        if (details[1].equals(n.getNickname()) && n.getTime() == Long.parseLong(details[2])) {
            if (details[1].equals(this.node.getNickname()) && this.node.getTime() == Long.parseLong(details[2])) {
                // If this is the node joining the room, remove itself from existing room and clear the messages
                room.clear();
                messages.clear();
                // Adjust hash map
                int currentVal = chatrooms.get(this.node.getcRoom());
                chatrooms.put(this.node.getcRoom(), currentVal - 1);
                int curVal = chatrooms.get(details[3]);
                chatrooms.put(details[3], curVal + 1);
                this.node.setcRoom(details[3]);
                System.out.println("Joined chatroom");
            } else {
                if (this.node.room.contains(n)) {
                    // Remove from the room if the joining node is leaving
                    room.remove(n);
                    n.setcRoom(details[3]);
                    int val = chatrooms.get(details[3]);
                    chatrooms.put(details[3], val + 1);
                    chatrooms.put(details[4], (this.node.chatrooms.get(details[4]) - 1));
                    System.out.println(details[1] + " has left the chatroom");
                } else {
                    int val = chatrooms.get(details[3]);
                    chatrooms.put(details[3], val + 1);
                    int val2 = chatrooms.get(details[4]);
                    chatrooms.put(details[4], val2 - 1);
                    n.setcRoom(details[3]);
                }
            }
        }
    }
```

```
// Check this is the node joining, if so then add all existing nodes in the room to their list of clients in the chatroom
if (details[1].equals(this.node.getNickname()) && this.node.getTime() == Long.parseLong(details[2])) {
    for (Node n2: tempList) {
        room.add(n2);
    }
    // Send a packet to those clients informing them this node is joining
    for (Node n3: room) {
        String roomInfo = "/ROOM@" + this.node.getNickname() + "@" + this.node.getTime();
        DatagramPacket rpi = new DatagramPacket(roomInfo.getBytes(), roomInfo.length(), n3.getcIp(), n3.getcPort());
        ds.send(rpi);
    }
    // Add itself to the list
    room.add(this.node);
}
// Handles information of nodes joining the chat room a node is in
} else if (details[0].equals("/ROOM")) {
    for (Node n: clients) {
        if (details[1].equals(n.getNickname()) && n.getTime() == Long.parseLong(details[2])) {
            room.add(n);
            String msgsToSync = "";
            for (Message m: messages) {
                msgsToSync += m.getmMessage() + ">" + m.getmSender() + ">" + m.getmTStamp() + ">" +m.getMessageID() + "~";
            }
            // Send joining node the room's chat history
            String msgPkt = "/MSG@" + msgsToSync;
            System.out.println("Synced: " + msgPkt);
            DatagramPacket msgDp = new DatagramPacket(msgPkt.getBytes(), msgPkt.length(), packet.getAddress(), packet.getPort());
            ds.send(msgDp);
            break;
        }
    }
```

- Viewing rooms
  - Nodes display their current room, members of the room and a list of other chatrooms stored in the hash map alongside the number of people in each hash map
    - This is updated when nodes join/create/leave rooms

```
// Allows the user to view chat rooms on the network and who is in their current room
} else if (input.equalsIgnoreCase("/viewrooms")) {
    System.out.println("---------------------------------------");
    System.out.println("+ Current Room: ");
    System.out.println(this.node.getcRoom());
    System.out.println("+ Users in room: ");
    for (Node inRoom: room) {
        System.out.println(inRoom.getNickname());
    }
    System.out.println("CHATROOM    # OF USERS");
    System.out.println("---------------------------------------");
    for (Map.Entry<String, Integer> set : chatrooms.entrySet()) {
        System.out.println(set.getKey() + "      " + set.getValue());
    }
    userCommand = true;
```

Recovering Lost Data

- Nodes are able to check for lost data and request it from nodes that have it
- Each node keeps track of how many message it sends in the messageNo variable
- Users can type '/syncdata' which will run a loop counting how many messages it has received from each node
- Each number is sent to the respective client who compares it against their messageNo variable
  - If the number is the same it returns a message letting the requester know that they are not missing any messages
  - Otherwise it will send its list of local messages back as a response
- Once the node receives a reply, it will either be satisfied as being synced if it receives a SYNCED response or will start to parse through the messages it received and check that each is present
- If a message is found to be missing it is added to a missing message list

- Once each message has been checked another for loop is run to compare the timestamp of each missing message and insert them into the right index of the chat history list
- I used multiple for loops here to try and keep the run time down to $O(n^2)$ as I was wary of packet loss if the loops took too long to process
- Packet loss can be simulated using the previously mentioned '/failsim' command to test this feature

```
// Send a request to sync data by counting the amount of messages each node has sent that is stored on the local chat history
// Sends it to each node to confirm the number is right
} else if (input.equalsIgnoreCase("/syncdata")) {
    for (Node c: room) {
        int count = 0;
        for (int i = 0; i < this.node.messages.size(); i++) {
            if (c.getNickname().equals(this.node.messages.get(i).getmSender())) {
                count++;
            }
        }
        if (count > 0) {
            String missingMsg = "/SYNCREQ@" + count;
            DatagramPacket drp = new DatagramPacket(missingMsg.getBytes(), missingMsg.length(), c.getcIp(), c.getcPort());
            ds.send(drp);
        }
    }
    userCommand = true;
```

```
// Allows a node to find missing data
} else if (details[0].equals("/SYNCREQ")) {
    // If the amount of messages sent by a node stored by the requester is the same as the number the node has sent
    if (Integer.parseInt(details[1]) == this.node.messageNo) {
        System.out.println("Synced");
        // If so reply that the node is synced
        String syncReply = "/SYNCREPLY@" + this.node.getNickname() + "@SYNCED";
        DatagramPacket drp = new DatagramPacket(syncReply.getBytes(), syncReply.length(), packet.getAddress(), packet.getPort());
        ds.send(drp);
    } else {
        System.out.println("Not Synced");
        // Otherwise send a list of its local chat history (messages sent by this node) back to the requester
        String localMsgs = "/SYNCREPLY@" + this.node.getNickname() + "@";
        for (Message msg: this.node.localMessages) {
            localMsgs += msg.getmSender() + ">" + msg.getmMessage() + ">" + msg.getmTStamp() + ">" + msg.getMessageID() + "~";
        }
        DatagramPacket drp = new DatagramPacket(localMsgs.getBytes(), localMsgs.length(), packet.getAddress(), packet.getPort());
        ds.send(drp);
    }
```

```
// Reply to the request for missing data
} else if (details[0].equals("/SYNCREPLY")) {
    ArrayList<String> missingMsgs = new ArrayList<String>();
    ArrayList<Message> senderMsgs = new ArrayList<Message>();
    System.out.println("Reply received");
    if (details[2].equals("SYNCED")) {
        System.out.println(details[1] + ": Messages Synced");
    } else {
        // If missing messages parse messages similarly to how they parsed in /MSG when adding chat history
        System.out.println(details[1] + ": Missing Messages");
        System.out.println("Syncing...");
        String[] nodeMsgs = details[2].split("~");
        // Create a list of messages sent by the sender stored on this node
        for (Message m: this.node.messages) {
            if (m.getmSender().equals(details[1])) {
                senderMsgs.add(m);
            }
        }
        // Check each missing message
        for (String s: nodeMsgs) {
            boolean missing = true;
            String[] msgDetails = s.split(">");
            // If this message is not found in the existing list then add it to the missing messages list
            for (Message m: senderMsgs) {
                if (m.getMessageID() == Integer.parseInt(msgDetails[3])) {
                    missing = false;
                }
            }
            if (missing) {
                missingMsgs.add(s);
            }
        }
        for (String mm: missingMsgs) {
            System.out.println(mm);
        }
        boolean inserted = false;

  for (String mm: missingMsgs) {
      String[] mmDetails = mm.split(">");
      // Now insert the missing messages back into the chat history at their correct points based on their time stamp
      for (int l = 0; l < this.node.messages.size(); l++) {
          if (Long.parseLong(mmDetails[2]) < messages.get(l).getmTStamp()) {
              if (inserted) {
                  continue;
              } else {
                  messages.add(l, new Message(mmDetails[1], mmDetails[0], Long.parseLong(mmDetails[2]), Integer.parseInt(mmDetails[3])));
                  inserted = true;
              }
          }
      }
      if (!inserted) {
          messages.add(new Message(mmDetails[1], mmDetails[0], Long.parseLong(mmDetails[2]), Integer.parseInt(mmDetails[3])));
      }
  }
  System.out.println("Synced");
```

Transfer Binary Files

- Text files can be transferred across the network to each node
- The user is prompted to give a file path and the program checks if the file exists
- If the file exists a scanner reads in each line and adds it to a string, with each line being divided by a symbol that can later be parsed similar to how messages are handled
- The nickname of the node sending the file is appended to the end and sent to all nodes
- Each node then checks if the file exists, prints a message if it does advising of this fact
- Otherwise, a new file is created with the name inputted by the sender and each line of the string sent in the message is parsed and written to the new file

```
// Send a text file across the network
}else if (input.equalsIgnoreCase("/sendfile")) {
    message += "/FILE@";
    System.out.println("");
    // Get file path
    System.out.println("");
    System.out.print("Enter full path of file: ");
    String input3 = sc.nextLine();
    message += input3 + "@";
    File f = new File(input3);
    // Check file exists
    if(f.exists() && !f.isDirectory()) {
        // Reads in each line and adds it to the message to be sent with the '>' as a delimiter to allow parsing on the other end
        BufferedReader br = new BufferedReader(new FileReader(f));
        String data = br.readLine();
        while(data!=null)
        {
            message += data + ">";  // Writing in the console
            data = br.readLine();
         }
        br.close();
    } else {
        System.out.println("Invalid file, try again");
    }
    message += "@" + this.node.cNick;
    System.out.println("Sent file");
    textMsg = true;
```

```
// Receive a file and download it
} else if (details[0].equals("/FILE")) {
    // Ensure node cannot send it to itself
    for (String s: details) {
        System.out.println(s);
    }
    if (details[3].equals(this.node.getNickname())) {
        break;
    } else {
        System.out.println("");
        System.out.println("Incoming file: " + details[1] + " from " + details[3]);
            try {
                // Create a new file and parse the incoming details into it
                File dwnld = new File(details[1]);
                if (dwnld.createNewFile()) {
                    FileWriter fileWrite = new FileWriter(details[1]);
                    String[] fileContents = details[2].split(">");
                    for (String s: fileContents) {
                        fileWrite.write(s);
                        fileWrite.write(System.getProperty( "line.separator" ));
                    }
                    fileWrite.close();
                    System.out.println("File downloaded: " + dwnld.getName());
                } else {
                    FileWriter fileWrite = new FileWriter(details[1]);
                    String[] fileContents = details[2].split(">");
                    for (String s: fileContents) {
                        fileWrite.write(s);
                        fileWrite.write(System.getProperty( "line.separator" ));
                    }
                    fileWrite.close();
                    System.out.println("File downloaded: " + dwnld.getName());
                }
            } catch (IOException e) {
                System.out.println("An error occurred.");
                e.printStackTrace();
            }
    }
```

Clearing Message List and Rebuilding From Network

- Nodes are able to clear their message list with '/rebuilddata'
- This sends a '/MSGREQ' packet with the node's nickname and timestamp to all other nodes in the room and clears the current chat history
- Receiving nodes will loop through clients in their room list and when it matches one to the requester's nickname and timestamp included in the packet it sends its chat history back, again as a single string with delimiters, to the node
- The node receives a packet with '/MSG' as the keyword at the start and begins to parse the string sent in the packet
    - Each message is delimited by a character and using String.split() each message is stored in an array
    - Each element of this array is then split again using a different delimiter to get each component of the message and a new message is created and stored in the chat history
    - If the chat history is empty each message is added sequentially as it is read
    - Otherwise each message's timestamp is compared with those already in the array and inserted where appropriate
- The node's status is then set to synced
- The terminal is cleared and the chat history printed out again

```
// Send a request for other nodes chat histories to rebuild it
} else if (input.equalsIgnoreCase("/rebuilddata")) {
    message = "/MSGREQ@" + this.node.getNickname() + "@" + this.node.getTime();
    this.node.messages.clear();
    this.node.setSynced(false);
```

```
// Handles requests for chat history/message syncing
} else if (details[0].equals("/MSGREQ")) {
    if (details[1].equals(this.node.getNickname()) && this.node.getTime() == Long.parseLong(details[2])) {
        continue;
    } else {
        // A node will send a request to all other nodes in the room for their chat histories, and when this is received they will send them ba
        for (Node c: room) {
            if (c.getNickname().equals(details[1]) && c.getTime() == Long.parseLong(details[2])) {
                String msgsToSync = "";
                for (Message m: messages) {
                    msgsToSync += m.getmMessage() + ">" + m.getmSender() + ">" + m.getmTStamp() + ">" +m.getMessageID() + "~";
                }
                String msgPkt = "/MSG@" + msgsToSync;
                System.out.println("Synced: " + msgPkt);
                DatagramPacket msgDp = new DatagramPacket(msgPkt.getBytes(), msgPkt.length(), c.getcIp(), c.getcPort());
                ds.send(msgDp);
            }
        }
    }

// Node parses incoming chat histories and adds to their chat history
} else if (details[0].equals("/MSG")) {
    if (!this.node.isSynced()) {
        this.node.setSynced(true);
        System.out.println("Syncing...");
        if (details.length > 1) {
            // Chat details are sent in 1 packet and parsed eg. <message1>~<message2>~<message3>
            String[] syncMsgs = details[1].split("~");
            if (this.node.messages.size() > 0) {
                int count = 0;
                for (count = 0; count < this.node.messages.size(); count++) {
                    // Each message that has been parsed is now further parsed into its components
                    // Eg <sender>@<message>@<timestamp>@messagenumber becomes sender | message | time stamp | message number
                    String[] syncDetails = syncMsgs[count].split(">");
                    // Check that this message is already in the history, if not add it
                    if (syncDetails[0].equals(this.node.messages.get(count).getmSender()) && syncDetails[1].equals(this.node.messages.get(count
                        count++;
                        continue;
                    } else {
                        // Create new message, add it to the list and print it out
                        Message newMsg = new Message(syncDetails[0], syncDetails[1], Long.parseLong(syncDetails[2]), Integer.parseInt(syncDetai
                        newMsg.setMessageID(Integer.parseInt(syncDetails[3]));
                        newMsg.setSynced(true);
                        //System.out.println(syncDetails[0] + "> " + syncDetails[1]);
                        messages.add(count, newMsg);
                        count++;
                    }
                }
```

```
            for (int i = count; i < syncMsgs.length; i++) {
                String[] syncDetails = syncMsgs[i].split(">");
                Message newMsg = new Message(syncDetails[0], syncDetails[1], Long.parseLong(syncDetails[2]), Integer.parseInt(syncDetails[3
                newMsg.setMessageID(Integer.parseInt(syncDetails[3]));
                newMsg.setSynced(true);
                //System.out.println(syncDetails[0] + "> " + syncDetails[1]);
                messages.add(newMsg);
            }
        } else {
            // If list is empty just add all messages without checking if they exist
            for (int i = 0; i < syncMsgs.length; i++) {
                String[] syncDetails = syncMsgs[i].split(">");
                Message newMsg = new Message(syncDetails[0], syncDetails[1], Long.parseLong(syncDetails[2]), Integer.parseInt(syncDetails[3
                newMsg.setMessageID(Integer.parseInt(syncDetails[3]));
                newMsg.setSynced(true);
                //System.out.println(syncDetails[0] + "> " + syncDetails[1]);
                messages.add(newMsg);
            }
        }
        System.out.print("\033[H\033[2J");
        System.out.flush();
        for (Message m: messages) {
            System.out.println(m.getmSender() + "> " + m.getmMessage());
        }
    }
}
```

Redraw Chat History

- Loops through the message list and prints out each one
- The list will already be in order from previous checks

```
// Redraw the chat history in order
} else if (input.equalsIgnoreCase("/redraw")) {
    this.node.redraw(this.node);
    userCommand = true;
```

```
// Print chat history in order
public void redraw(Node lnode) {
    System.out.print("\033[H\033[2J");
    System.out.flush();
    for (int i = 0; i < lnode.messages.size(); i++) {
        System.out.println(lnode.messages.get(i).getmSender() + "> " + lnode.messages.get(i).getmMessage());
    }
    System.out.println("Chat Refreshed");
}
```

External Interface

- User calls this function with ‘/readtitle’ and inputs a website URL
- If the URL is invalid it will be caught with a MalformedURL exception in the try/catch block
- The program uses a BufferedReader and URL object to read in the HTML of the web page
- This is stored in a string
- Using substring multiple times to manipulate the string, the title of the web page is read in and printed out

```
// Reads in and parses a web page and outputs the title of the web page
} else if (input.equalsIgnoreCase("/readtitle")) {
    try {
        System.out.println("");
        System.out.print("Enter URL: ");
        String urlin = sc.nextLine();
        URL url = new URL(urlin);
        BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));
        String addResult;
        String result = "";
        while ((addResult = in.readLine()) != null) {
            result += addResult;
        }
        result = result.substring(result.indexOf("<title") + 7);
        result = result.substring(result.indexOf(">") + 1);
        result = result.substring(0, result.indexOf("</title>"));
        System.out.println("Webpage title: ");
        System.out.println(result);
    } catch (MalformedURLException ex) {
        System.out.println("Enter valid url");
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    userCommand = true;
```

Miscellaneous Features

- Typing '/list' will print a list of commands and what they do

```
} else if (input.equalsIgnoreCase("/list")) {
    System.out.println("/quit       Leave the chat");
    System.out.println("/nickname   Set a new username");
    System.out.println("/robot      Starts automated chat bot to generate dialogue");
    System.out.println("/joinroom   Join a new chatroom");
    System.out.println("/createroom Create a new chatroom");
    System.out.println("/viewrooms  View a list of existing chatrooms");
    System.out.println("/sendfile   Send a file to other nodes in your chatroom");
    System.out.println("/redraw     Reprint the chat in order");
    System.out.println("/rebuilddata    Clear local data and rebuild from network");
    System.out.println("/syncdata   Check for missing data");
    userCommand = true;
```

- Program is started from StartNode, which generates the local node and then runs the start method containing all the thread starts

```
import java.io.IOException;

public class StartNode {
    public static void main(String[] args) throws IOException
    {
        InetAddress lIp = InetAddress.getByName("localhost");
        Random r = new Random();
        int low = 1;
        int high = 100000000;
        String nickN = "" + r.nextInt(high-low) + low;
        if (args.length == 1) {
            Node uServer = new Node(lIp, nickN, args[0]);
            uServer.Start(uServer);
        } else if (args.length > 1){
            System.out.println("Please specify a chatroom to join (with no spaces) or leave blank for default");
        } else {
            Node uServer = new Node(lIp, nickN);
            uServer.Start(uServer);
        }
    }
}
```