Chat and VoIP Project

Eleni Koumparidou
Ioannis Dellios

December 2024

Contents

1 Introduction

This project involves the development of an end-to-end chat and VoIP application using the UDP Protocol, implemented in Java. This application is designed to enable users to communicate using only their IP addresses and logical ports, which are managed locally by the application. The graphical user interface (GUI) for the project was provided by Professor Miltiadis Siavvas as part of the Computer Networks 2 course.

2 Text Chat

2.1 Text Chat Interface

When the app window is open, users can send and receive text messages that appear in a dedicated text message area within the window. Each message in this area is labeled to indicate whether it was sent by the user `(Local)` or received from the other person `(Remote)`. Users can send messages by typing into the white input box below the message area and click the `Send` button.

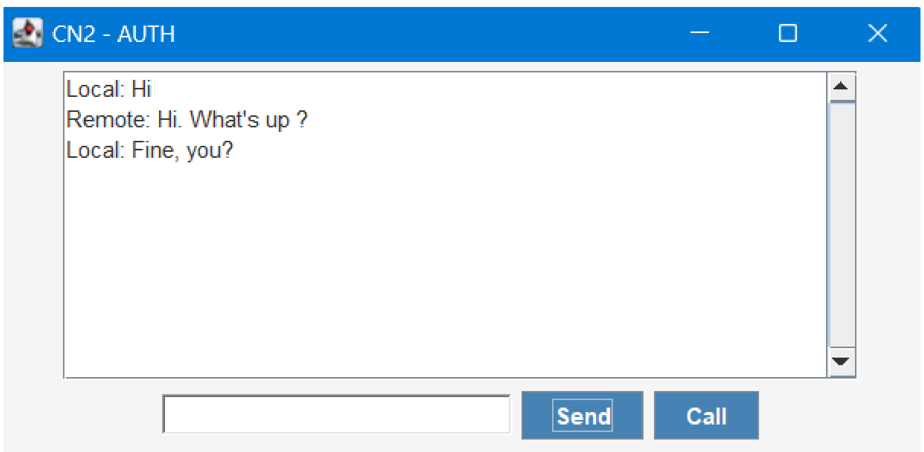

Figure 1: Chat Interface

2.2 Text Chat Implementation

2.2.1 Sending Messages

When `Send` button is clicked, `actionPerformed()` function is called, invoking `sendMessages()` to handle text messages sending. This function checks if there is an available message to send and a remote address (line 3), which the user can adjust inside `main()`. A DatagramPacket containing the message is created with the size of 1024 bytes and directed to the available remote address and port.

Then it is sent through the designed socket for text messages, `textSocket`. After sending, the message appends to the text area of the window with `Local` label in the sender's interface, and the text input box clears (lines 14 and 15). Any errors that might occur are caught by catch statement and printed in the text area.

```java
1  public void sendMessages() {
2      String sendMessage = inputTextField.getText();
3      if (sendMessage.isEmpty() || remoteAddress == null) {
4          textArea.append("Error: No remote address or
               message is empty.\n");
5          return;
6      }
7      try {
8          // send message
9          byte[] buffer = new byte[1024];
10         buffer = sendMessage.getBytes();
11         DatagramPacket packet = new DatagramPacket(buffer,
               buffer.length, remoteAddress, textRemotePort);
12         textSocket.send(packet);
13
14         // show message in sender's window
15         textArea.append("Local: " + sendMessage + "\n");
16         inputTextField.setText("");
17     } catch (IOException ex) {
18         ex.printStackTrace();
19         textArea.append("Error sending message: " +
               ex.getMessage() + "\n");
20     }
21 }
```

Listing 1: Sending messages function

2.2.2 Receiving Messages

A thread for the `receiveMessages()` function is created within `main()` to make sure that receiving messages is constantly available. Inside a forever while loop, `receiveMessages()` creates `DatagramPacket` objects of 1024 bytes to receive data sent through the `textSocket`. The received text messages are represented by a `String` variable, `receiveMess`, which is initialized from the packets received every time and later printed in the text area. To ensure communication is properly terminated, when the chat window is closed, it is important that the thread responsible for receiving messages has stopped receiving data from `textSocket` before closing the texting socket (lines 19-26).

```java
public static void main(String[] args) {
// Initialization code here
// ...

// Receive text messages
    new Thread(() -> app.receiveMessages()).start();
}
```

Listing 2: Thread initiallization

```java
public void receiveMessages() {
    while (true) {
        try {
            // Receive message
            byte[] buffer = new byte[1024];
            DatagramPacket packet = new
                DatagramPacket(buffer, buffer.length);
            textSocket.receive(packet);
            String receiveMess = new
                String(packet.getData(), 0,
                packet.getLength());

            // Show message
            textArea.append("Remote: " + receiveMess +
                "\n");
        } catch (IOException ex) {
            if (!textSocket.isClosed()) { // Error when
                socket is active
                ex.printStackTrace();
                textArea.append("Error receiving message: "
                    + ex.getMessage() + "\n");
            }
        }
    }
}
```

Listing 3: Function for receiving messages

2.3 Packet Capture of Text Messages

Some text messages exchanged through the application were captured as packets using Wireshark, as shown in Figure 2. At the top of the figure information about the delivered packets is provided, including remote and local IP Addresses and ports, the protocol used and the message length. At the bottom, the selected message is displayed in two formats: on the left, its hexadecimal representation, and on the right, the actual text message.

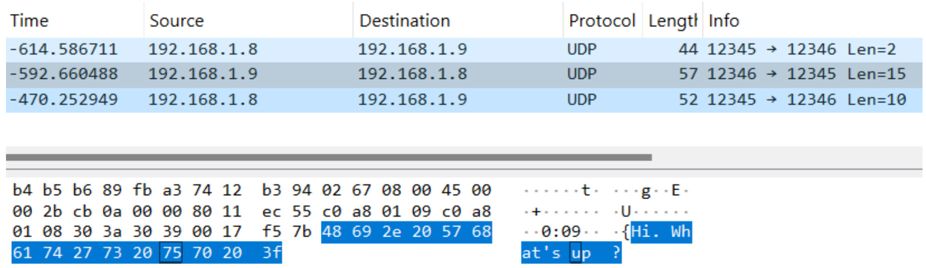

Figure 2: Text packets exchange, Wireshark

2.4 Maximum Transmission Unit

We expect that the Maximum Transmission Unit (MTU) using the UDP protocol is approximately 1480 bytes. In order to confirm that, we set the buffer sizes for sending and receiving text messages to 3000 bytes and then we sent a message of 2128 bytes in total. The entire text message was successfully delivered, but we observed with use of Wireshark that it was fragmented into two parts. The first fragment was 1480 bytes and the second fragment was 648 bytes, as shown in Figure 3, confirming the expected MTU.

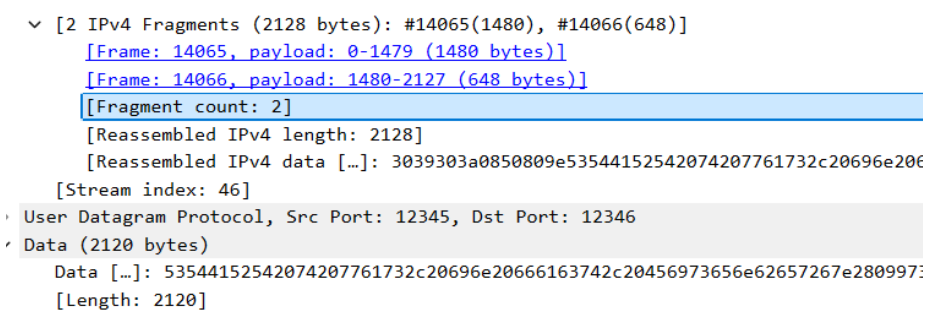

Figure 3: MTU testing, Wireshark

3 Voice Call

Voice call between users can proceed by clicking the `Call` button. A function called `handleCallButton()` is created to manage actions happening every time `Call` button is clicked. The first time the button is clicked, `Start calling...` is displayed in the text area and the blue button labeled `Call` turns into a green button labeled `End Call`. When call ends, it returns to the initial state.

3.1 Voice Call Implementation

Once `Call` button is clicked, `call()` is invoked. The `call()` function consists of a thread to record and send voice packets of 1024 bytes, `sendVoiceThread`, and a while loop to receive and listen to incoming voice packets. Both actions are activated only when `isCalling` variable is `true`, which indicates that the call is in progress. When `Call` button is clicked again, `isCalling` becomes `false` from `handleCallButton()` and call ends using `endCall()` for closing open lines used for recording and listening `(voiceLine, speakerLine)`.

The audio format used was PCM (pulse code modulation) with the following specifications: (i) sampling rate of 8000 samples/sec, (ii) 8-bit sample size, (iii) single channel, and (iv) signed samples.

```java
public void call() {
// Initialization code here
//.....
    // Thread for sending voice packets
    sendVoiceThread = new Thread(() -> {
        while (isCalling) {
            try {
                // Record and send voice packets
                byte[] buffer = new byte[1024];
                int audioMessage = voiceLine.read(buffer, 0, buffer.length);
                DatagramPacket voiceSend = new DatagramPacket(buffer, AudioMessage, remoteAddress, voiceRemotePort);
                voiceSocket.send(voiceSend);

            } catch (IOException ex) {
                if (!voiceSocket.isClosed()) {
                    ex.printStackTrace();
                    textArea.append("Error in sending voice: " + ex.getMessage() + "\n");
                }
            }
        }
    });
    sendVoiceThread.start();

    // Receive and listen to voice packets
    while (isCalling) {
        try {
            byte[] buffer = new byte[1024];
              DatagramPacket voiceReceive = new DatagramPacket(buffer, buffer.length);
            voiceSocket.receive(voiceReceive);
            speakerLine.write(voiceReceive.getData(), 0, voiceReceive.getLength());
```

```java

        } catch (IOException ex) {
            if (!voiceSocket.isClosed()) {
                ex.printStackTrace();
                textArea.append("Error in receiving voice:
                    " + ex.getMessage() + "\n");
            }
        }
    }
//....
}
```

Listing 4: Voice call function

3.2 Packet Capture of Voice Messages

Figure 4 displays some voice packets captured in Wireshark during a voice call through the application. Important information about the delivered packets is provided, including remote and local IP Addresses and ports, the protocol used and the message length (1024 bytes each).

No.	Time	Source	Destination	Protocol	Lengtl	Info
206	34.251703	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
208	34.472979	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
209	34.472979	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
210	34.563949	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
211	34.564342	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
212	34.564722	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
213	34.565186	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
215	34.609238	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
216	34.816664	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
217	34.928984	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
218	34.942194	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
219	35.069066	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
220	35.070441	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
221	35.070441	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
222	35.194877	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
223	35.249413	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
224	35.313071	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024
225	35.321482	192.168.1.167	192.168.1.150	UDP	1066	12350 → 12351 Len=1024
226	35.439006	192.168.1.150	192.168.1.167	UDP	1066	12351 → 12350 Len=1024

Figure 4: Voice packets exchange, Wireshark

In Figure 5 there is an analysis of a voice packet, in which the bytes of the packet is shown in binary form.

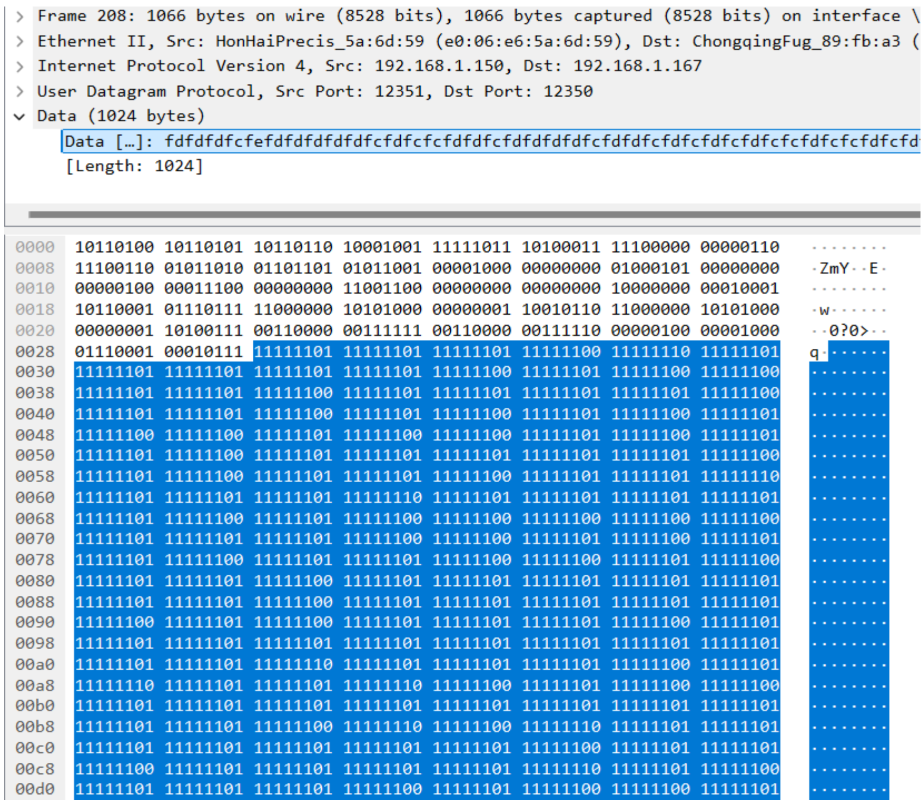

Figure 5: Payload analysis of a voice packet, Wireshark