P2P Chat and VoIP Application using UDP in Java

Aristotle University of Thessaloniki - Department of Electrical and Computer Engineering

Computer Networks II

Epameinondas Bakoulas and Maria Sotiria Kostomanolaki

December 2024

Abstract

This report presents the development of a Peer-to-Peer (P2P) Chat and Voice over IP (VoIP) application, created as part of the Computer Networks II course at Aristotle University of Thessaloniki. The application uses Java's `java.net` library for network communications, providing hands-on experience with real-time data exchange and concurrency. Designed for communication between two peers, it supports instant messaging and multimedia data transfer.

The project also explores the concept of switching between UDP and TCP protocols to understand their trade-offs. Although implementing the switch to TCP did not result in a functional feature, this exploration deepened our understanding of Internet Protocols, challenges posed by NAT (Network Address Translation), and the role of port forwarding in enabling P2P communication. Additionally, cryptographic techniques were integrated to secure the exchanged data, emphasizing the importance of privacy in P2P communication.

1 UDP Chat and VoIP Application

1.1 Variables

The application uses two `DatagramSocket` objects:

- `messageSocket` for handling message communication
- `voiceSocket` for handling voice data

This separation is important to avoid conflicts between the different types of data (text and voice) that are transmitted over UDP, as each socket is dedicated to a specific purpose. Additionally, the application uses four `ports`:

- **Local Ports**: `LOCAL_PORT_MESSAGE (12345)` is used for receiving messages, and `LOCAL_PORT_VOICE (12346)` is used for receiving voice data. Each type of communication (messages and voice) requires a dedicated port to **listen** for incoming data.
- **Remote Ports**: `REMOTE_PORT_MESSAGE (12345)` is used for sending messages to the remote peer, and `REMOTE_PORT_VOICE (12346)` is used for sending voice data. These ports ensure that data is **sent** to the appropriate destination, depending on whether it is a message or voice.

This setup enables efficient, organized handling of different data streams (text vs. voice) and ensures that there are no interference or data delivery issues for each type of communication.

1.2 Initialization Process and Socket Management

The application ensures efficient resource management and smooth communication by dynamically handling socket initialization. Below is an itemized explanation of the initialization process:

1. **Default UDP Initialization:**
   - Method Used: `initUDPSockets()`

- When Used: On app startup or when the user switches to UDP via the protocol switch button.
- What It Does: Creates and binds UDP sockets for messaging and voice communication using predefined local ports. This allows the app to start communication immediately using the UDP protocol.

2. **Releasing Resources:**

   - Methods Used: `deinitUDPSockets()` and `deinitTCPSockets()`
   - When Used: Before switching to a different protocol.
   - What It Does: Ensures that sockets from the inactive protocol are properly closed, freeing up the associated resources and avoiding conflicts on the same ports.

This modular approach minimizes resource usage, prevents port conflicts, and allows seamless protocol switching without restarting the application.

2 Encryption

The application uses the `AES` (Advanced Encryption Standard) algorithm to secure data exchanged between peers. The encryption key is hardcoded within the application and is used for both encrypting and decrypting messages. For optimal security, the key should be exchanged between peers securely to maintain the privacy and integrity of the communication.

While the implementation does not currently include a secure key exchange mechanism, methods such as the `Diffie-Hellman` key exchange protocol can be employed to securely share encryption keys between peers in a real-world scenario. This is an important consideration for ensuring private communication over potentially insecure networks.

Additionally, an example of the encrypted payload produced by the application can be seen in Section 4, Figures 3 and 4, which illustrate how data is transformed through encryption. This highlights the practical application of the encryption process within the system.

3 Fullstack Application

Using the Java framework `Spring Boot` for the backend and the frontend library `React`, we developed a full-stack application. Each backend instance is designed to communicate exclusively with a single frontend instance, ensuring secure and end-to-end communication.

The backend exposes its functionality through REST APIs, which the frontend consumes using the `Axios` library. This architecture facilitates efficient and structured interaction between the client and server, enabling seamless real-time data exchange.

To run the application, both the backend and frontend servers need to be started:

- **Backend Server**: Run the command `mvn spring-boot:run`.
- **Frontend Server**: Run the command `npm run dev`.

For users who prefer not to launch the full-stack application, an alternative standalone GUI application can be run by executing the `App.java` file. This approach bypasses the need for server setups.

The core files of the full-stack application are:

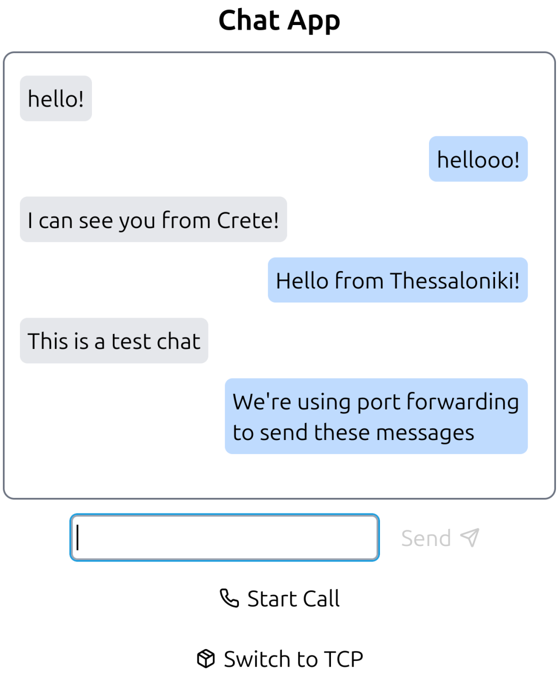
- `AppController.java`: Handles backend logic and API endpoints.
- `App.jsx`: Implements the main frontend component.

The chat application user interface can be seen in Figure 1.

4 Wireshark packets

4.1 Port Forwarding

To test the functionality of our application in a remote setting, we utilized **port forwarding** by creating a custom rule in the NAT and Security settings of our personal Wi-Fi routers. This configuration allowed us to redirect incoming traffic to the appropriate device within our local networks.



Chat App

hello!

hellooo!

I can see you from Crete!

Hello from Thessaloniki!

This is a test chat

We're using port forwarding to send these messages

Figure 1: Chat Application User Interface

In addition to setting up port forwarding, testing required temporarily disabling the firewall on both devices to ensure that the network traffic was not blocked. After exchanging our public IP addresses, we configured the code accordingly by:

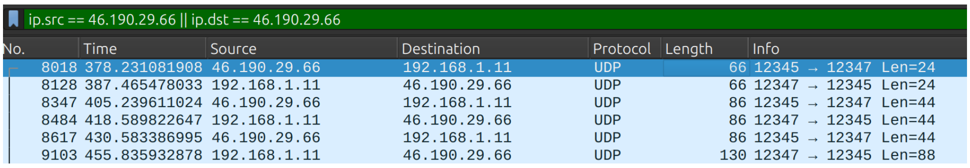
- Setting the `REMOTE_IP` variable to the IP address of the other peer.
- Adjusting the number of ports used for communication to match the settings configured in the port forwarding rules.

With these configurations in place, we successfully tested the application's functionality remotely, exchanging messages between two devices across different networks.

4.2 UDP Messages Packets

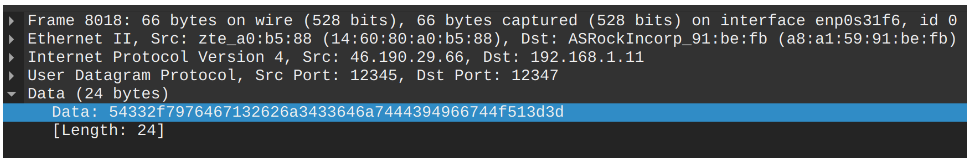
After successfully establishing communication between our two devices and exchanging several messages, we used **Wireshark** to analyze the network traffic and demonstrate the functionality of the application. The messages exchanged are encrypted using the `AES` algorithm, ensuring the privacy of the communication. The encryption key used for this process is `123456789ABCDEFG`.

In Figure 2, we observe the packets being sent and received between the two devices, identified by their distinct IP addresses. This confirms the proper transmission of data between peers. Figures 3 and 4 further illustrate the encryption in action by showing the encrypted payload of the messages. These encrypted messages highlight the security measures implemented in the application, ensuring that sensitive data remains private and inaccessible to unauthorized parties.



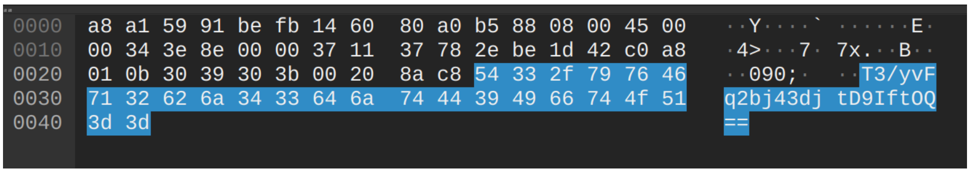
```
ip.src == 46.190.29.66 || ip.dst == 46.190.29.66
No.    Time          Source         Destination    Protocol Length Info
  8018 378.231081908 46.190.29.66   192.168.1.11   UDP          66 12345 → 12347 Len=24
  8128 387.465478033 192.168.1.11   46.190.29.66   UDP          66 12347 → 12345 Len=24
  8347 405.239611024 46.190.29.66   192.168.1.11   UDP          86 12345 → 12347 Len=44
  8484 418.589822647 192.168.1.11   46.190.29.66   UDP          86 12347 → 12345 Len=44
  8617 430.583386995 46.190.29.66   192.168.1.11   UDP          86 12345 → 12347 Len=44
  9103 455.835932878 192.168.1.11   46.190.29.66   UDP         130 12347 → 12345 Len=88
```

Figure 2: UDP message packet exchanging with port forwarding



```
▸ Frame 8018: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface enp0s31f6, id 0
▸ Ethernet II, Src: zte_a0:b5:88 (14:60:80:a0:b5:88), Dst: ASRockIncorp_91:be:fb (a8:a1:59:91:be:fb)
▸ Internet Protocol Version 4, Src: 46.190.29.66, Dst: 192.168.1.11
▸ User Datagram Protocol, Src Port: 12345, Dst Port: 12347
▾ Data (24 bytes)
    Data: 54332f7976467132626a3433646a7444394966744f513d3d
    [Length: 24]
```

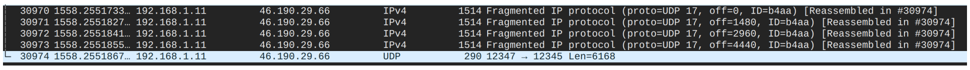
Figure 3: UDP message packet (encrypted Payload marked)



```
0000  a8 a1 59 91 be fb 14 60  80 a0 b5 88 08 00 45 00   ··Y····` ······E·
0010  00 34 3e 8e 00 00 37 11  37 78 2e be 1d 42 c0 a8   ·4>···7· 7x.··B··
0020  01 0b 30 39 30 3b 00 20  8a c8 54 33 2f 79 76 46   ··090;·  ··T3/yvF
0030  71 32 62 6a 34 33 64 6a  74 44 39 49 66 74 4f 51   q2bj43dj tD9IftOQ
0040  3d 3d                                              ==
```

Figure 4: UDP message packet (encrypted Payload marked)

We can also observe that messages larger than the Maximum Transmission Unit (MTU) size of UDP packets are fragmented and reassembled on the receiver, as shown in Figure 5. This process of fragmentation and reassembly is inherent to UDP communication when transmitting data that exceeds the MTU size.

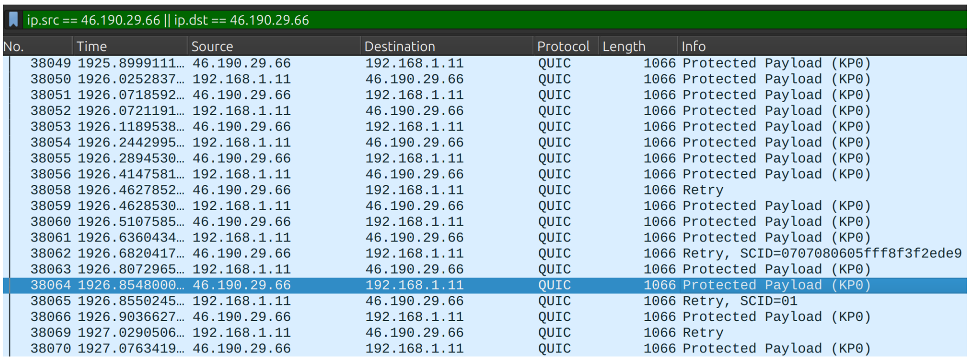


```
  30970 1558.2551733… 192.168.1.11    46.190.29.66    IPv4    1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=b4aa) [Reassembled in #30974]
  30971 1558.2551827… 192.168.1.11    46.190.29.66    IPv4    1514 Fragmented IP protocol (proto=UDP 17, off=1480, ID=b4aa) [Reassembled in #30974]
  30972 1558.2551841… 192.168.1.11    46.190.29.66    IPv4    1514 Fragmented IP protocol (proto=UDP 17, off=2960, ID=b4aa) [Reassembled in #30974]
  30973 1558.2551855… 192.168.1.11    46.190.29.66    IPv4    1514 Fragmented IP protocol (proto=UDP 17, off=4440, ID=b4aa) [Reassembled in #30974]
└ 30974 1558.2551867… 192.168.1.11    46.190.29.66    UDP      290 12347 → 12345 Len=6168
```

Figure 5: Large UDP message fragmented

4.3 UDP Voice Packets

The application supports the exchange of voice data, enabling a form of voice calling between peers. Once both peers activate this feature by pressing the call button, the application establishes a continuous exchange of voice packets (as shown in Figure 6).



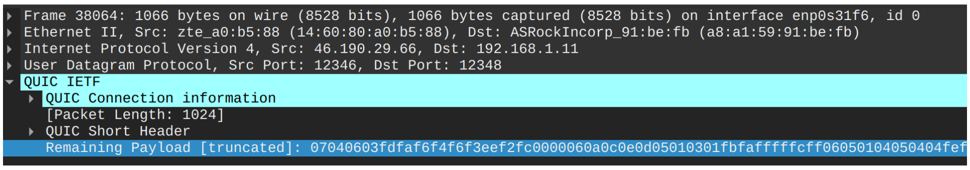
```
ip.src == 46.190.29.66 || ip.dst == 46.190.29.66
No.   Time          Source        Destination   Protocol Length Info
38049 1925.8999111… 46.190.29.66  192.168.1.11  QUIC     1066 Protected Payload (KP0)
38050 1926.0252837… 192.168.1.11  46.190.29.66  QUIC     1066 Protected Payload (KP0)
38051 1926.0718592… 46.190.29.66  192.168.1.11  QUIC     1066 Protected Payload (KP0)
38052 1926.0721191… 192.168.1.11  46.190.29.66  QUIC     1066 Protected Payload (KP0)
38053 1926.1189538… 46.190.29.66  192.168.1.11  QUIC     1066 Protected Payload (KP0)
38054 1926.2442995… 192.168.1.11  46.190.29.66  QUIC     1066 Protected Payload (KP0)
38055 1926.2894530… 46.190.29.66  192.168.1.11  QUIC     1066 Protected Payload (KP0)
38056 1926.4147581… 192.168.1.11  46.190.29.66  QUIC     1066 Protected Payload (KP0)
38058 1926.4627852… 46.190.29.66  192.168.1.11  QUIC     1066 Retry
38059 1926.4628530… 192.168.1.11  46.190.29.66  QUIC     1066 Protected Payload (KP0)
38060 1926.5107585… 46.190.29.66  192.168.1.11  QUIC     1066 Protected Payload (KP0)
38061 1926.6360434… 192.168.1.11  46.190.29.66  QUIC     1066 Protected Payload (KP0)
38062 1926.6820417… 46.190.29.66  192.168.1.11  QUIC     1066 Retry, SCID=0707080605fff8f3f2ede9
38063 1926.8072965… 192.168.1.11  46.190.29.66  QUIC     1066 Protected Payload (KP0)
38064 1926.8548000… 46.190.29.66  192.168.1.11  QUIC     1066 Protected Payload (KP0)
38065 1926.8550245… 192.168.1.11  46.190.29.66  QUIC     1066 Retry, SCID=01
38066 1926.9036627… 46.190.29.66  192.168.1.11  QUIC     1066 Protected Payload (KP0)
38069 1927.0290506… 192.168.1.11  46.190.29.66  QUIC     1066 Retry
38070 1927.0763419… 46.190.29.66  192.168.1.11  QUIC     1066 Protected Payload (KP0)
```

Figure 6: UDP continuous voice packets

The voice packets are sent in a continuous stream and stored on the receiving end using a 1024-byte buffer. This buffer temporarily holds the incoming voice data, which is then played back to the user through the `SourceDataLine` class. The continuous exchange of voice packets stops when the user presses the call button again to end the call. Figures 7 and 8 provide a detailed view of the payload of these voice packets, showcasing the structure and data content exchanged during the voice call.

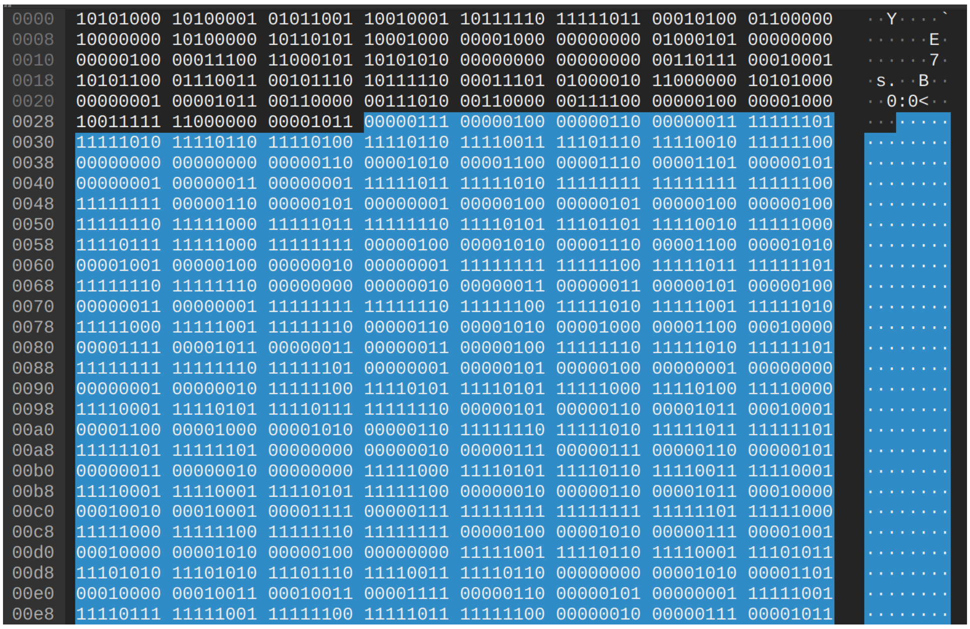
5 TCP Implementation and Protocol Switching

In this project, we explored the implementation of a TCP connection as an alternative to the existing UDP-based communication. To achieve this, we used the following sockets on both sides of the communication:



```
Frame 38064: 1066 bytes on wire (8528 bits), 1066 bytes captured (8528 bits) on interface enp0s31f6, id 0
Ethernet II, Src: zte_a0:b5:88 (14:60:80:a0:b5:88), Dst: ASRockIncorp_91:be:fb (a8:a1:59:91:be:fb)
Internet Protocol Version 4, Src: 46.190.29.66, Dst: 192.168.1.11
User Datagram Protocol, Src Port: 12346, Dst Port: 12348
QUIC IETF
  QUIC Connection information
  [Packet Length: 1024]
  QUIC Short Header
  Remaining Payload [truncated]: 07040603fdfaf6f4f6f3eef2fc0000060a0c0e0d05010301fbfaffffcff06050104050404fef
```

Figure 7: UDP voice packet (Payload marked)



```
0000  10101000 10100001 01011001 10010001 10111110 11111011 00010100 01100000   ··Y····`
0008  10000000 10100000 10110101 10001000 00001000 00000000 01000101 00000000   ······E·
0010  00000100 00011100 11000101 10101010 00000000 00000000 00110111 00010001   ······7·
0018  10101100 01110011 00101110 10111110 00011101 01000010 11000000 10101000   ·s.··B··
0020  00000001 00001011 00110000 00111010 00110000 00111100 00000100 00001000   ··0:0<··
0028  10011111 11000000 00001011 00000111 00000100 00000110 00000011 11111101   ········
0030  11111010 11110110 11110100 11110110 11110011 11101110 11110010 11111100   ········
0038  00000000 00000000 00000110 00001010 00001100 00001110 00001101 00000101   ········
0040  00000001 00000011 00000001 11111011 11111010 11111111 11111111 11111100   ········
0048  11111111 00000110 00000101 00000001 00000100 00000101 00000100 00000100   ········
0050  11111110 11111000 11111011 11111110 11110101 11101101 11110010 11111000   ········
0058  11110111 11111000 11111111 00000100 00001010 00001110 00001100 00001010   ········
0060  00001001 00000100 00000010 00000001 11111111 11111100 11111011 11111101   ········
0068  11111110 11111110 00000000 00000010 00000011 00000011 00000101 00000100   ········
0070  00000011 00000001 11111111 11111110 11111100 11111010 11111001 11111010   ········
0078  11111000 11111001 11111110 00000110 00001010 00001000 00001100 00010000   ········
0080  00001111 00001011 00000011 00000011 00000100 11111110 11111010 11111101   ········
0088  11111111 11111110 11111101 00000001 00000101 00000100 00000001 00000000   ········
0090  00000001 00000010 11111100 11110101 11110101 11111000 11110100 11110000   ········
0098  11110001 11110101 11110111 11111110 00000101 00000110 00001011 00010001   ········
00a0  00001100 00001000 00001010 00000110 11111110 11111010 11111011 11111101   ········
00a8  11111101 11111101 00000000 00000010 00000111 00000111 00000110 00000101   ········
00b0  00000011 00000010 00000000 11111000 11110101 11110110 11110011 11110001   ········
00b8  11110001 11110001 11110101 11111100 00000010 00000110 00001011 00010000   ········
00c0  00010010 00010001 00001111 00000111 11111111 11111111 11111101 11111000   ········
00c8  11111000 11111100 11111110 11111111 00000100 00001010 00000111 00001001   ········
00d0  00010000 00001010 00000100 00000000 11111001 11110110 11110001 11101011   ········
00d8  11101010 11101010 11101110 11110011 11110110 00000000 00001010 00001101   ········
00e0  00010000 00010011 00010011 00001111 00000110 00000101 00000001 11111001   ········
00e8  11110111 11111001 11111100 11111011 11111100 00000010 00000111 00001011   ········
```

Figure 8: UDP voice packet (Payload marked)

- `tcpMessageServerSocket`: Server-side TCP socket for messaging.
- `tcpMessageSocket`: Client-side TCP socket for messaging.
- `tcpVoiceServerSocket`: Server-side TCP socket for voice.
- `tcpVoiceSocket`: Client-side TCP socket for voice.

To integrate protocol switching into the application, we added a protocol switch button to the user interface. This button was designed to allow users to toggle between UDP and TCP protocols for both messaging and voice functionalities. However, despite our efforts, we were unable to overcome the challenges posed by P2P communication and successfully establish a TCP connection. These challenges were primarily related to the barriers introduced by Network Address Translation (NAT) and other network constraints.

Although the TCP implementation did not result in a functional connection, this exploration deepened our understanding of IP protocols and the complexities involved in establishing P2P communication using TCP. The reader is encouraged to review the code to gain further insights into the attempted implementation and the obstacles encountered.

6 References

- GitHub Repository: `https://github.com/NontasBak/CN2_AUTH_ChatAndVoIP`