# 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.

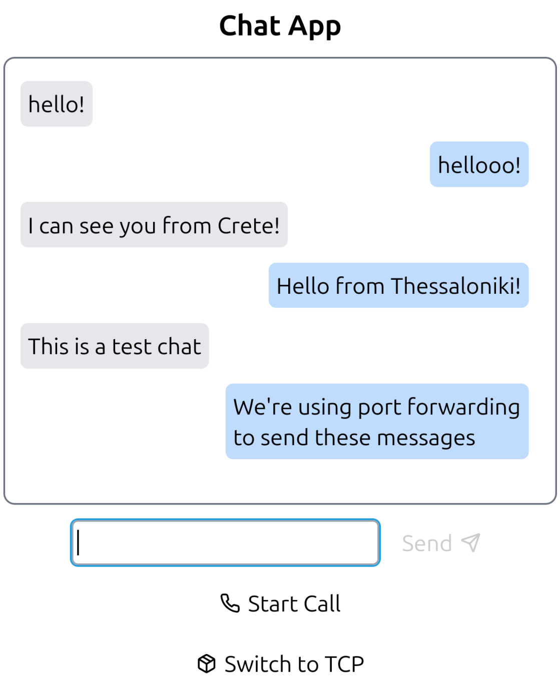


Figure 1: Chat Application User Interface

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

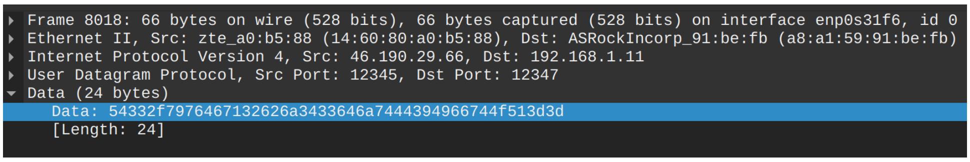


Figure 3: UDP message packet (encrypted Payload marked)

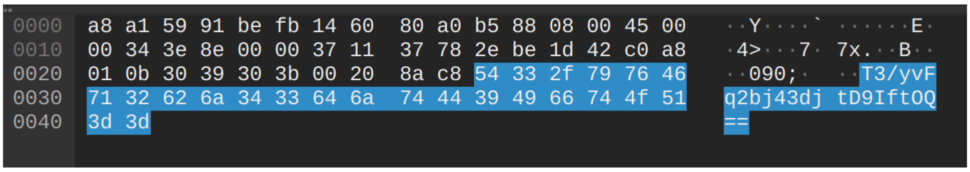


Figure 4: UDP message packet (encrypted Payload marked)

An important observation is that sending packets larger than 1024 bytes (after encryption) will not be successfully received by the other peer. This limitation arises because the application uses a fixed buffer size of 1024 bytes for processing incoming data. To handle larger packets, either the buffer size must be increased, or the message must be split into smaller packets that fit within the buffer size.

This phenomenon is illustrated in Figure 5, where we see the message being fragmented due to its size exceeding the buffer limit.

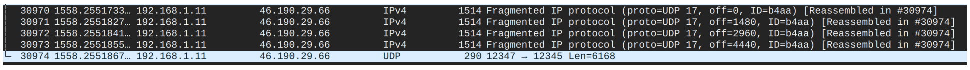


Figure 5: Large UDP message fragmented

## 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).

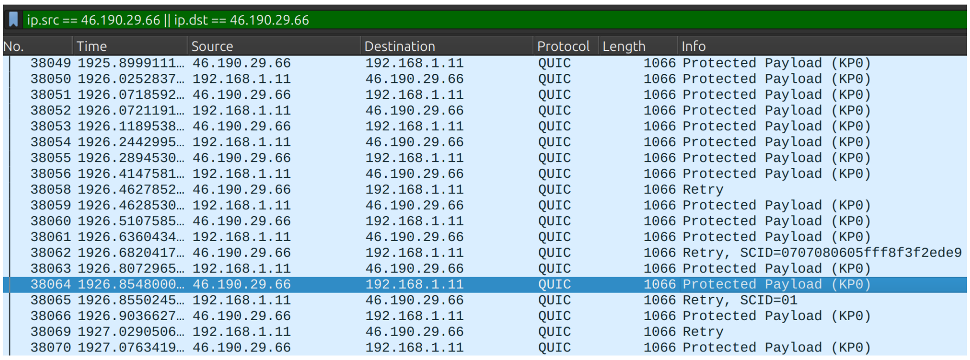


Figure 6: UDP continuous voice packets

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.

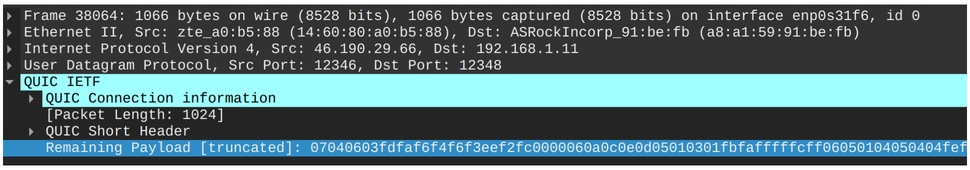


Figure 7: UDP voice packet (Payload marked)

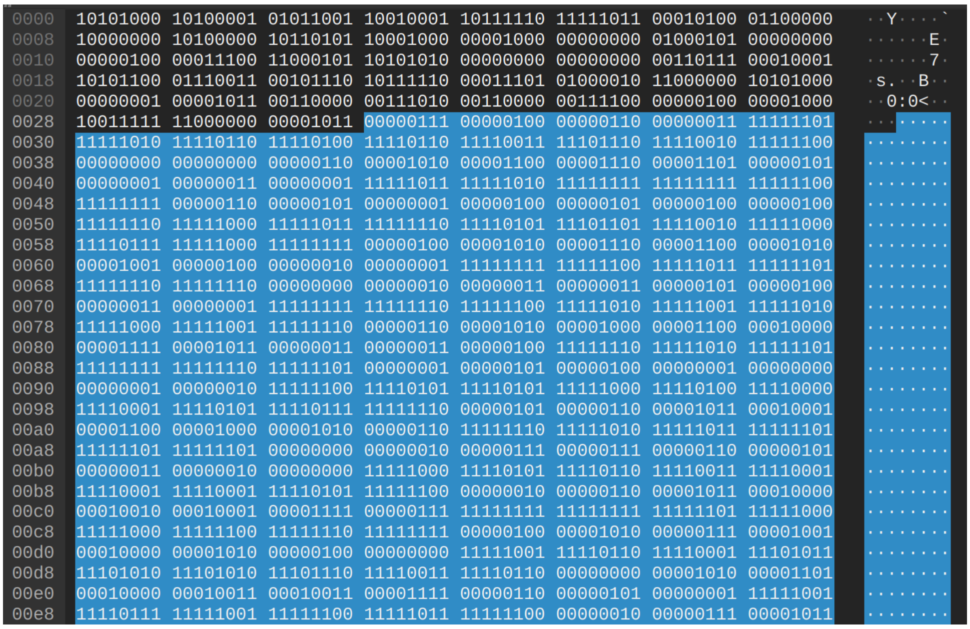


Figure 8: UDP voice packet (Payload marked)

## 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:

- `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`