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 is built using Java's `java.net` library to manage network communications.

The project demonstrates a deeper understanding of Internet Protocols (IP) by allowing users to switch between UDP and TCP protocols through a command. This feature highlights the trade-offs between speed and reliability in network communications. Additionally, cryptographic techniques are integrated to secure data exchanges, ensuring the privacy and integrity of communications.

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. **Switching to TCP:**
    - Methods Used: `initTCPSockets()`
    - When Used: When the user switches to TCP via the protocol switch button.
    - What It Does: Creates TCP server sockets for listening and establishes client connections for messaging and voice communication.
3. **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` encryption algorithm to secure the data exchanged between peers. The encryption key is hardcoded in the application and is used to encrypt and decrypt the messages. The key should be exchanged between the two peers securely to ensure that the communication is private and secure. There are methods like the `Diffie-Hellman` key exchange that can be used to securely exchange the encryption key between the peers, which is not implemented here but it's worth noting.

3 Fullstack Application

Using the Java Framework `Spring Boot` and the frontend library `React` we created a fullstack application. Each backend is allowed to communicate with a single frontend, ensuring that the communication is end-to-end. The backend services are exposed via REST APIs, which are consumed by the frontend using the `Axios` library. This setup allows for efficient and organized communication between the client and server, ensuring that data is exchanged seamlessly and in real-time.

Running the application requires starting both the backend and frontend servers. The backend server is started using the `mvn spring-boot:run` command, while the frontend server is started using the `npm run dev` command.

If someone doesn't want to run the fullstack application, they can run the `App.java` file that displays the GUI application. The two main files of the fullstack application are `AppController.java` and `App.jsx`.

4 Wireshark packets

4.1 UDP Messages Packets

We can see that the message is encrypted using the `AES` algorithm, which ensures the privacy of the communication. The key used is `123456789ABCDEFG`.

Sending packets that are larger than 1024 bytes (encrypted) will not be received by the other peer, since the application only uses a 1024 byte buffer. To send larger packets, the buffer size should be increased, or we need to split the message into smaller packets.

4.2 UDP Voice Packets

The voice packets are continuously sent and received between the two peers. The voice packets are sent in a continuous stream, and the application uses a 1024 byte buffer to store the incoming voice data. The buffer is then played back to the user using the `SourceDataLine` class.

5 References

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

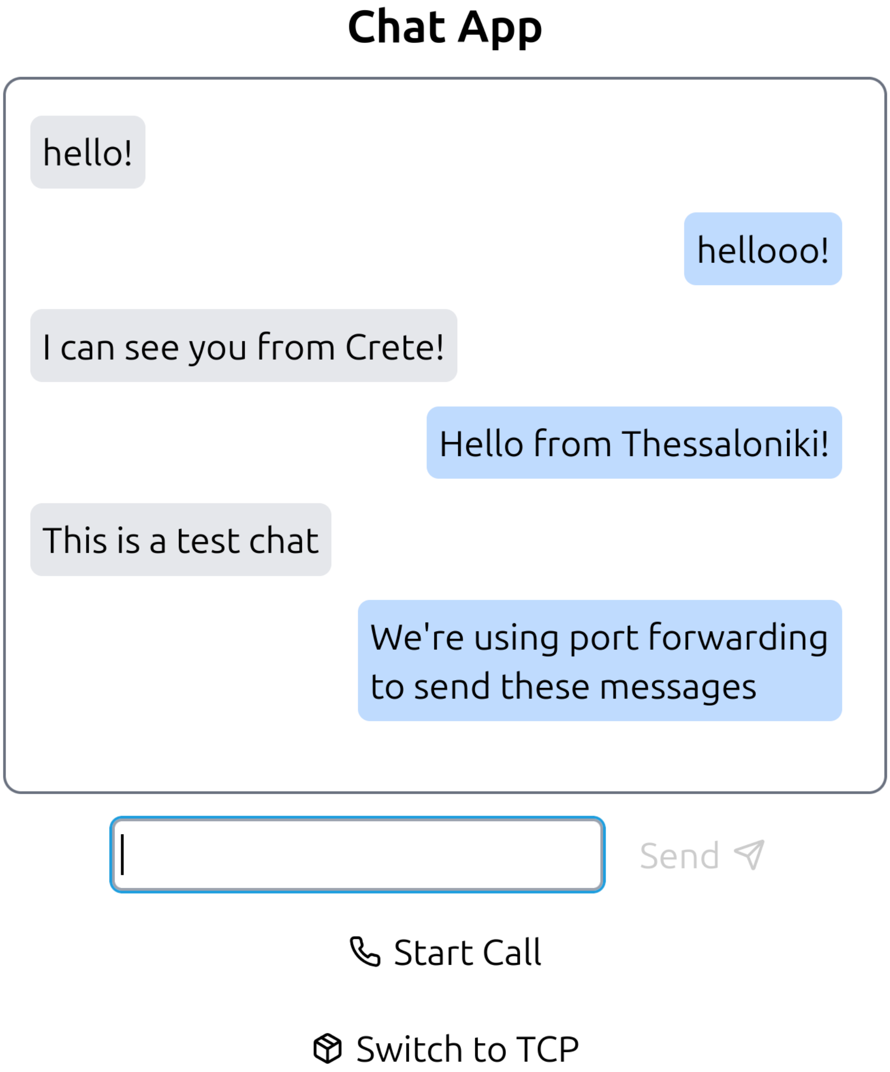

Figure 1: Chat Application User Interface

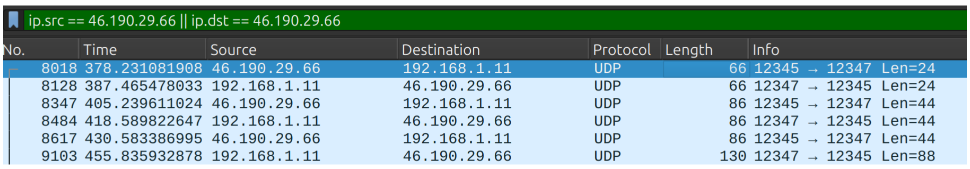

Figure 2: UDP message packet exchanging with port forwarding

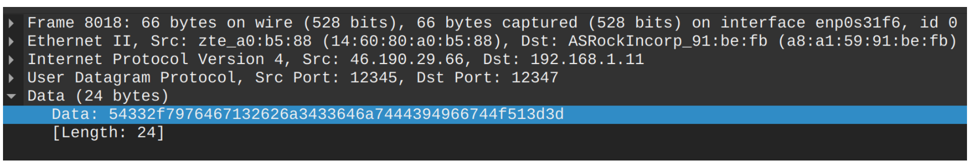

Figure 3: UDP message packet (encrypted Payload marked)

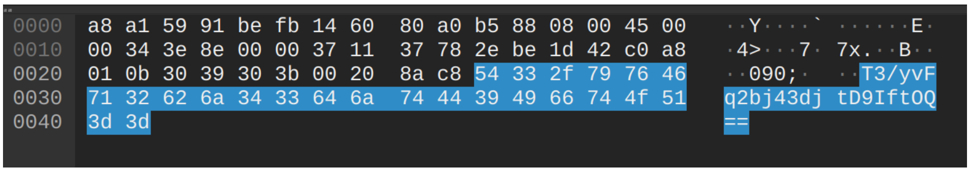

Figure 4: UDP message packet (encrypted Payload marked)

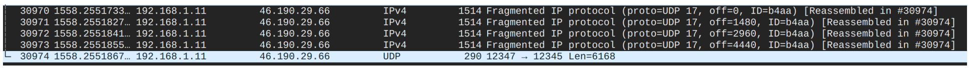

Figure 5: Large UDP message fragmented

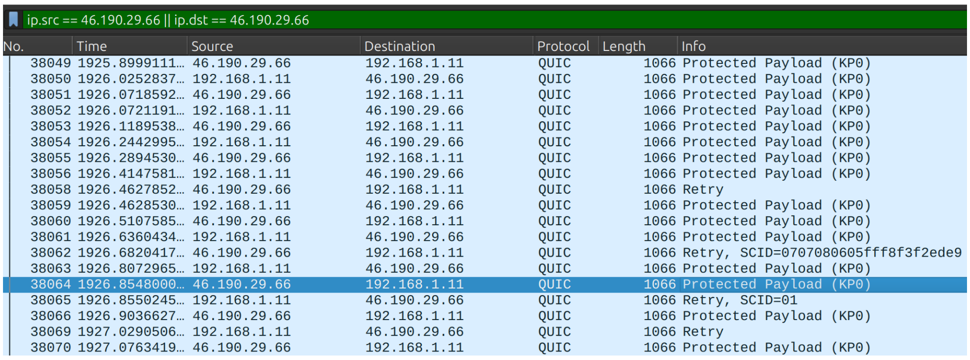

Figure 6: UDP continuous voice packets

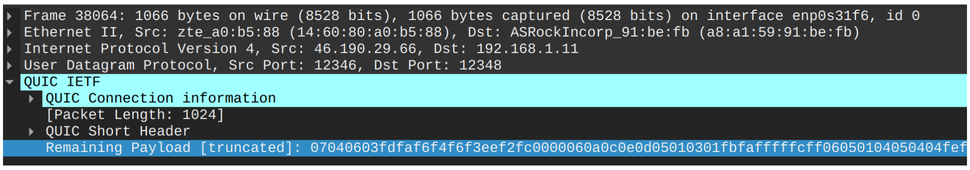

Figure 7: UDP voice packet (Payload marked)

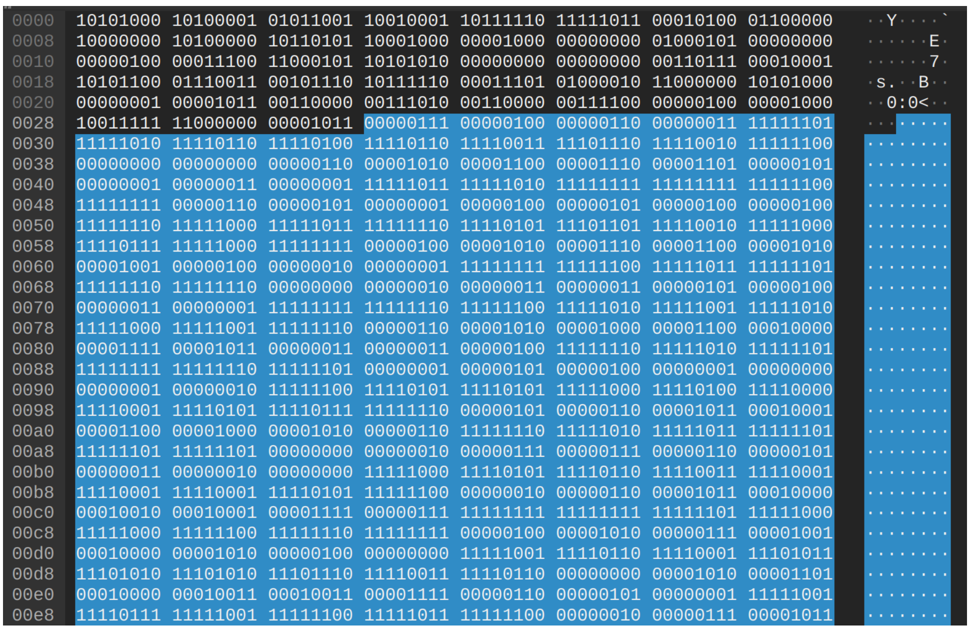

Figure 8: UDP voice packet (Payload marked)