

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 `123456789ABCDEFGH`.

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

Chat App

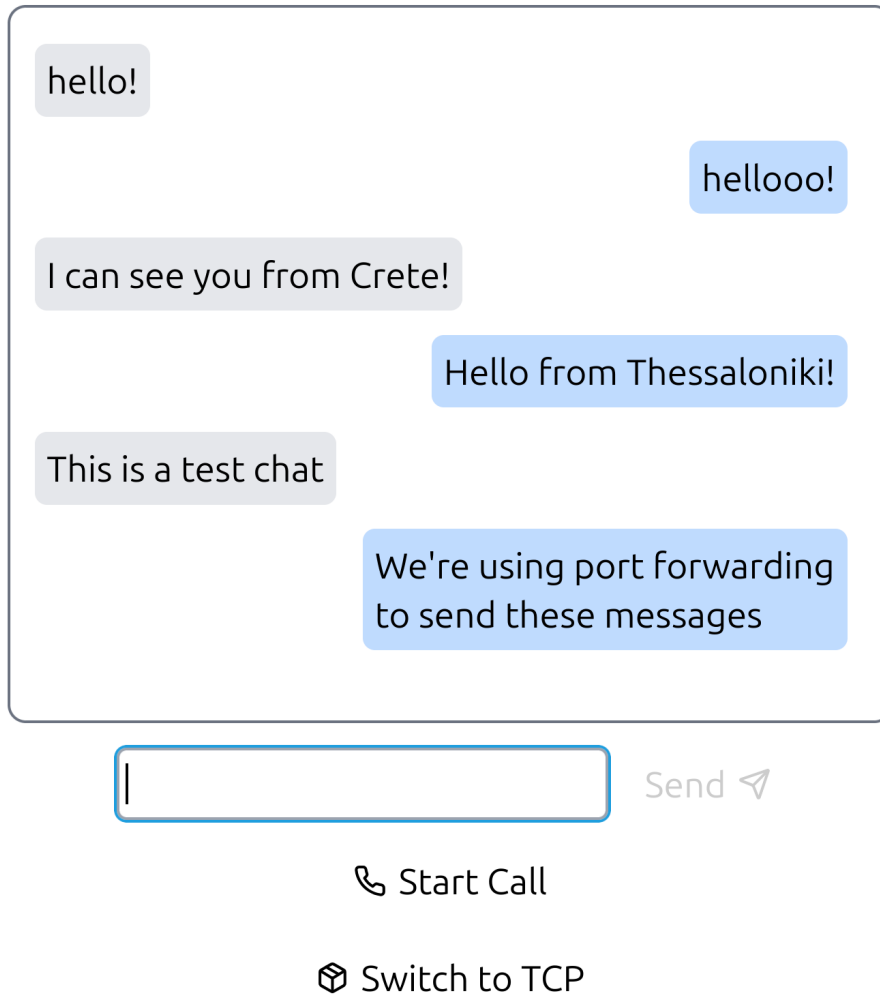


Figure 1: Chat Application User Interface

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 td9Ift0Q
0040 3d 3d ==

```

Figure 4: UDP message packet (encrypted Payload marked)

```

38970 1558.2551733... 192.168.1.11 46.190.29.66 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=b4aa) [Reassembled in #38974]
38971 1558.2551827... 192.168.1.11 46.190.29.66 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=1480, ID=b4aa) [Reassembled in #38974]
38972 1558.2551841... 192.168.1.11 46.190.29.66 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=2960, ID=b4aa) [Reassembled in #38974]
38973 1558.2551955... 192.168.1.11 46.190.29.66 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=4440, ID=b4aa) [Reassembled in #38974]
38974 1558.2551867... 192.168.1.11 46.190.29.66 UDP 290 12347 - 12345 Len=6168

```

Figure 5: Large UDP message fragmented

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=0707080605ffff8f3f2ede9
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

```

> 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 TETF
    > QUIC Connection information
    [Packet Length: 1024]
    > QUIC Short Header
    Remaining Payload [truncated]: 07040603fdaf6f4f6f3eef2fc0000060a0c0e0d05010301fbafffffccff06050104050404fe

```

Figure 7: UDP voice packet (Payload marked)

```

0000 10101000 10100001 01011001 10010001 10111110 11111011 00010100 01100000 ..Y....E.
0008 10000000 10100000 10110101 10001000 00001000 00000000 01000101 00000000 .....7.
0016 00000100 00011100 11000101 10101010 00000000 00000000 00110111 00010001 ..s...B.
0024 10101100 01110011 00101110 10111110 00011101 01000010 11000000 10101000 ..0:<...
0032 00000001 00001011 00110000 00111010 00110000 00111100 00000100 00001000 ..0:0<...
0040 10011111 11000000 00001011 00000111 00000100 00000110 00000011 11111101 .....
0048 11111010 11110110 11110100 11110110 11110011 11101110 11110010 11110010 .....
0056 00000000 00000000 00000110 00001010 00001100 00001110 00001101 00000101 .....
0064 00000001 00000011 00000001 11111011 11111011 11111111 11111111 11111100 .....
0072 11111111 00000110 00000101 00000001 00000100 00000101 00000100 00000100 .....
0080 11111110 11111110 00000000 00000010 00000011 00000011 00000101 00000100 .....
0088 00000011 00000001 11111111 11111110 11111100 11111010 11111001 11111010 .....
0096 00000001 00000010 11111100 11110101 11110101 11111000 11110100 11110000 .....
0104 11110001 11110101 11110111 11111110 00000101 00000110 00001011 00010001 .....
0112 00001100 00001000 00001010 00000110 11111110 11111010 11111011 11111101 .....
0120 11111101 11111101 00000000 00000010 00000111 00000111 00000110 00000101 .....
0128 00000011 00000010 00000000 11111000 11110101 11110110 11110011 11110001 .....
0136 11110001 11110001 11110101 11111100 00000010 00000110 00001011 00010000 .....
0144 00010010 00010001 00001111 00000111 11111111 11111111 11111101 11111000 .....
0152 11110000 11111100 11111110 11111111 00000100 00001010 00000111 00001001 .....
0160 00010000 00001010 00000100 00000000 11111001 11110110 11110001 11101011 .....
0168 11101010 11101010 11101110 11110011 11110110 00000000 00001010 00001101 .....
0176 00010000 00010011 00010011 00001111 00000110 00000101 00000001 11111001 .....
0184 11110111 11111001 11111100 11111011 11111100 00000010 00000111 00001011 .....

```

Figure 8: UDP voice packet (Payload marked)