TP 2 - Java for networks

UDP/TCP Chat App with Java sockets

Students:

CHEN Hong
GONÇALVES Sabrina

November 12, 2025

Objective

Develop a Client-Server Chat application, in UDP and TCP. Also, implement basic reliability features over UDP to understand its connectionless nature.

1 Creating a UDP Client-Server

1.1 UDP Server

In this part we implemented the `UDPServer` class that uses a `DatagramSocket` to receive messages and displays them on the console, with the client's address and port. The code includes a constructor to define the listening port, a `launch()` method to start the server, and a `main()` method that allows us to launch it with a simple command. The server runs in an infinite loop to handle multiple messages without interruption. Unlike TCP servers, this UDP version does not establish or maintain a connection, it receives each message independently, making it a lightweight and connectionless communication model.

To verify that the server was working correctly, we tested it using the `netcat` command. We could see the server printing the received text and the client's IP address, confirming that the server was correctly receiving UDP datagrams, as shown in Figure 1.

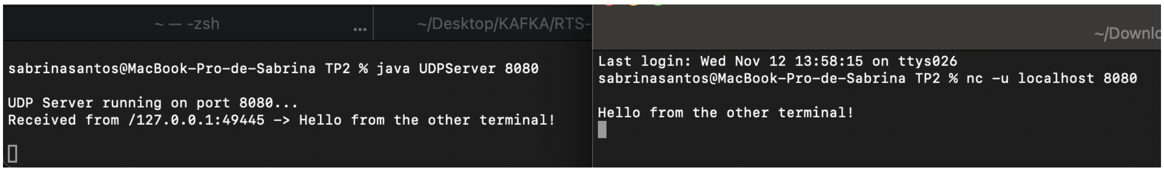


Figure 1: Testing the UDP server

> The program used until here is inside the folder Part 1 and it is saved as `UDPClient.java` and `UDPServer.java`.

Running the Part 1 code with the truncated analysis, as shown in Figure 2, we used the following command to send 1200 characters "A":

```
python -c "print('A'*1200)" | java UDPClient localhost 8080
```

The server received and displayed only the first 1024 bytes, showing a truncation warning—confirming that messages longer than 1024 bytes are correctly truncated as required.

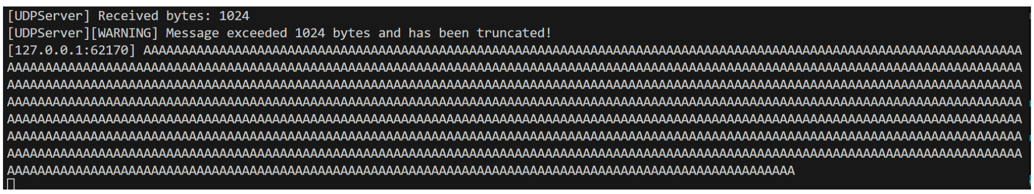


Figure 2: Testing the 1024-byte truncation on the UDP server

> The program used for the truncated analysis is inside the folder Part 1, inside the folder "Part1 with truncation check".

1.2 UDP Client

Now we create the `UDPClient` class to send messages directly from Java. The code is quite similar in structure but it performs the opposite role in the communication process. Instead of waiting to receive datagrams, the client actively sends them. It uses the DatagramSocket to create and send packets, and a console object to read text lines entered. Each message is converted into bytes using UTF-8 encoding and transmitted to the server's IP address and port.

Unlike the server, which runs continuously to listen for messages, the client operates interactively. It means that it waits for user to input, sends the message, and can immediately send another. So one class acts as a passive listener, while this one functions as an active sender.

When we ran both programs simultaneously, the client allowed us to type messages that were instantly displayed by the server in real time, as shown in Figure 3.

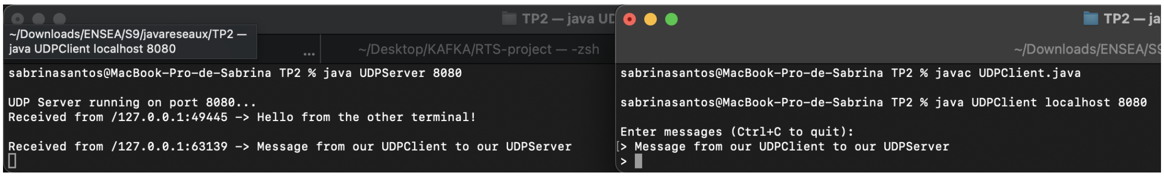
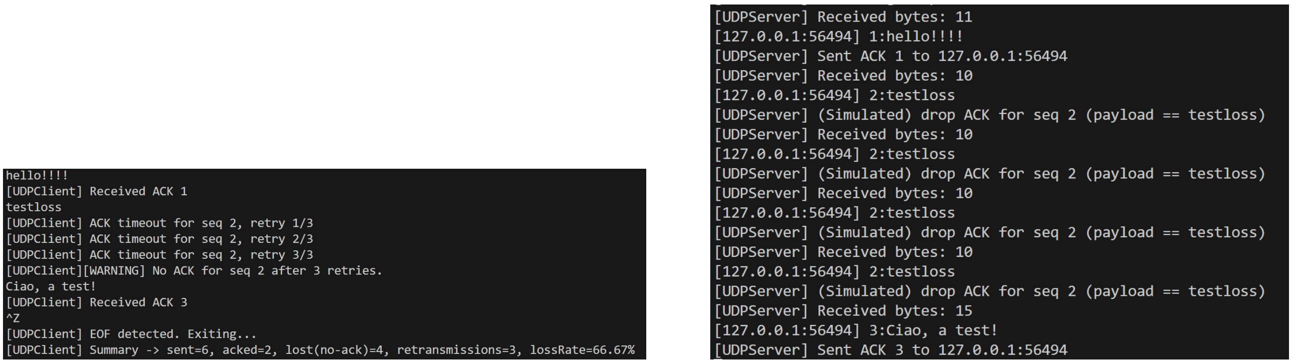


Figure 3: UDP client sending messages to the server

2 Additionals

2.1 UDP Reliability Challenges

To enhance the basic UDP client-server model by adding reliability features such as sequence numbering, acknowledgments (ACK), retransmission, and packet loss measurement.



(a) Reliable UDP client (b) Reliable UDP server

Figure 4: Reliable UDP setup: (a) client; (b) server

As shown in Figure 4, each client message was labeled `<seq>:<data>`, the server replied with ACK `<seq>`, and the client retried up to three times if no ACK arrived within 1 s. When the server intentionally dropped ACKs for messages containing testloss, the client correctly triggered timeouts and retransmissions.

At the end of the communication, the client summarized the transmission statistics, including the number of sent packets, ACKs received, retransmissions, and the calculated packet loss rate.

2.2 Datagram Packet Analysis

The code is documented with inline comments. Three buffers were used: `smallBuffer` (64 bytes), `largeBuffer` (2048 bytes), and `hugeBuffer` (65507 bytes).

To illustrate the results, we ran three short tests.

We tested and figured that the UDP protocol has a maximum payload of 65,507 bytes and does not guarantee message integrity when the receiver's buffer is smaller than the datagram. Oversized messages are rejected by the operating system, and truncation occurs when the buffer

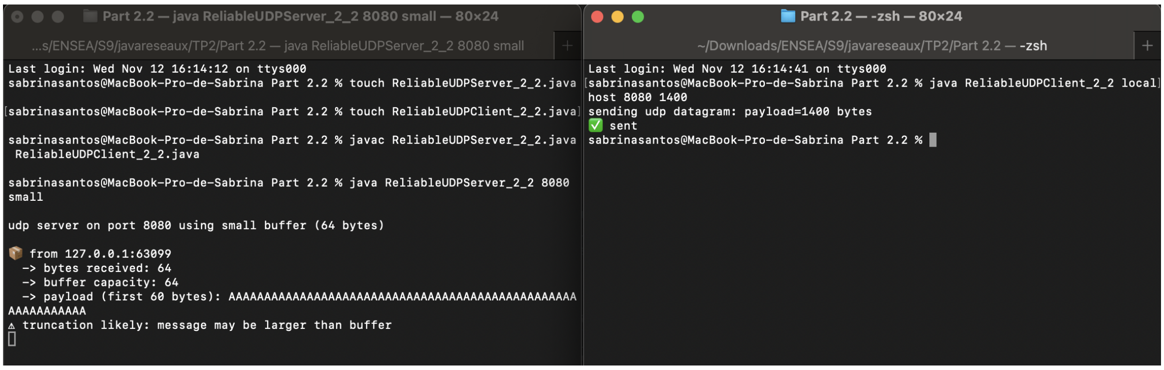


Figure 5: 64 byte buffer showing truncation of a 1400 byte message

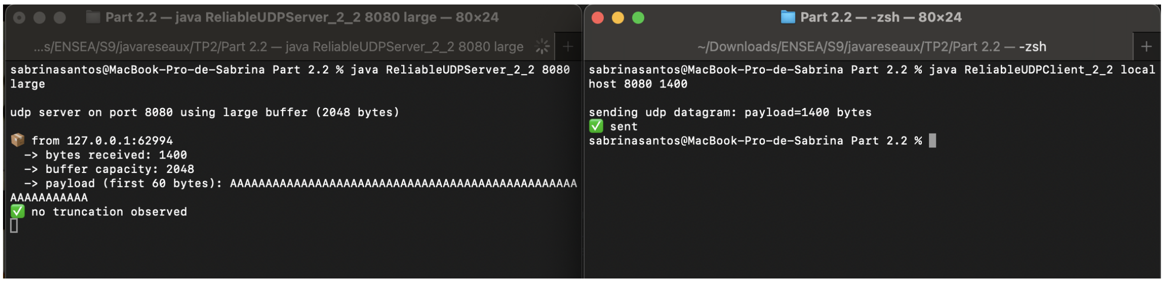


Figure 6: 2048 byte buffer receiving the full message

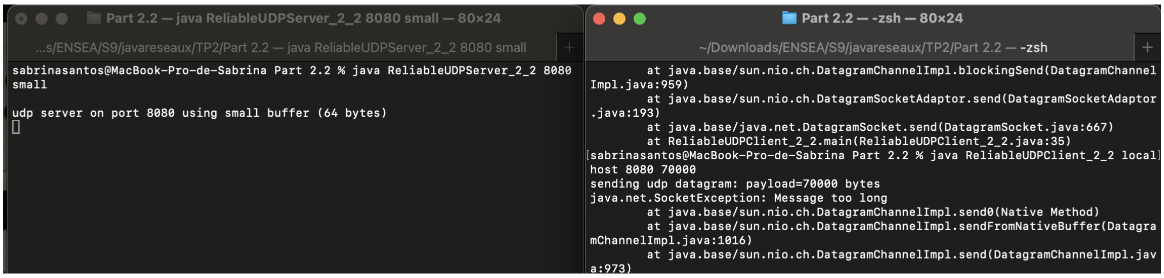


Figure 7: Message too long

is too small. Correct UDP communication requires properly configured buffer sizes on both ends and message fragmentation logic if larger data must be transmitted.

2.3 Connectionless Communication Demo

Using a simple client and server two short tests were performed.

In the first one, a client sent a message before the server was started, and no output appeared on the server side, confirming that UDP does not queue or guarantee delivery of datagrams (Figure 8).

In the second test, three different clients sent messages simultaneously to the same server, which received all of them instantly, independently and without any established session and no handshake required, as shown in Figure 9. UDP communication is fully connectionless. Also, the server doesn't track client state, just receive the information but does not keep it anywhere.

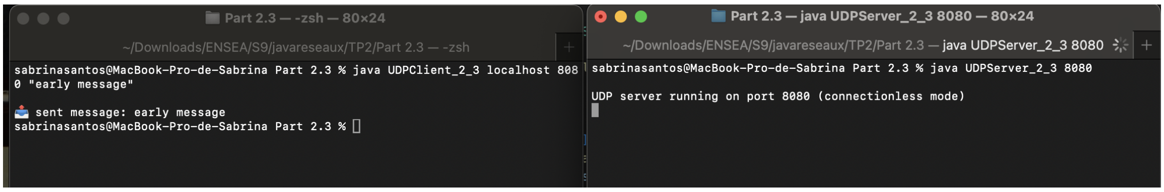


Figure 8: Client sending a message before the server starts

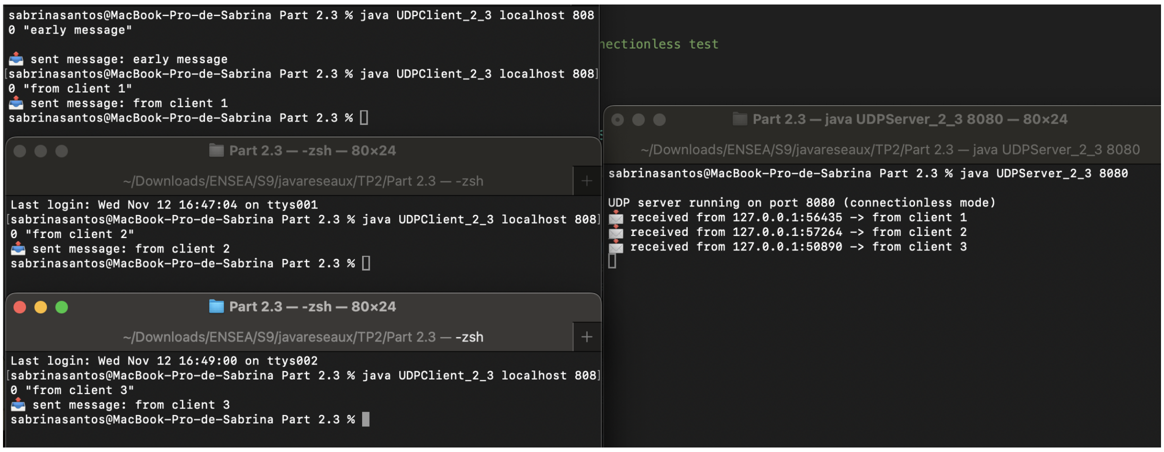


Figure 9: Server receiving messages from three independent clients

2.4 UDP Multicast Exploration

In this last part, we implemented a simple UDP multicast communication using two programs, a MulticastReceiver and a MulticastSender. The receiver joins a multicast group (`224.0.0.3`) using a `MulticastSocket` and listens on a given port, while the sender transmits a message to the same group and port.

During testing, we found that multicast loopback is blocked by default on macOS, preventing local message delivery even though both receivers successfully joined the multicast group, as shown in Figure 10. However, when the same code was executed on a Windows machine, both receivers received the message simultaneously, confirming the expected multicast behavior (Figure 11).

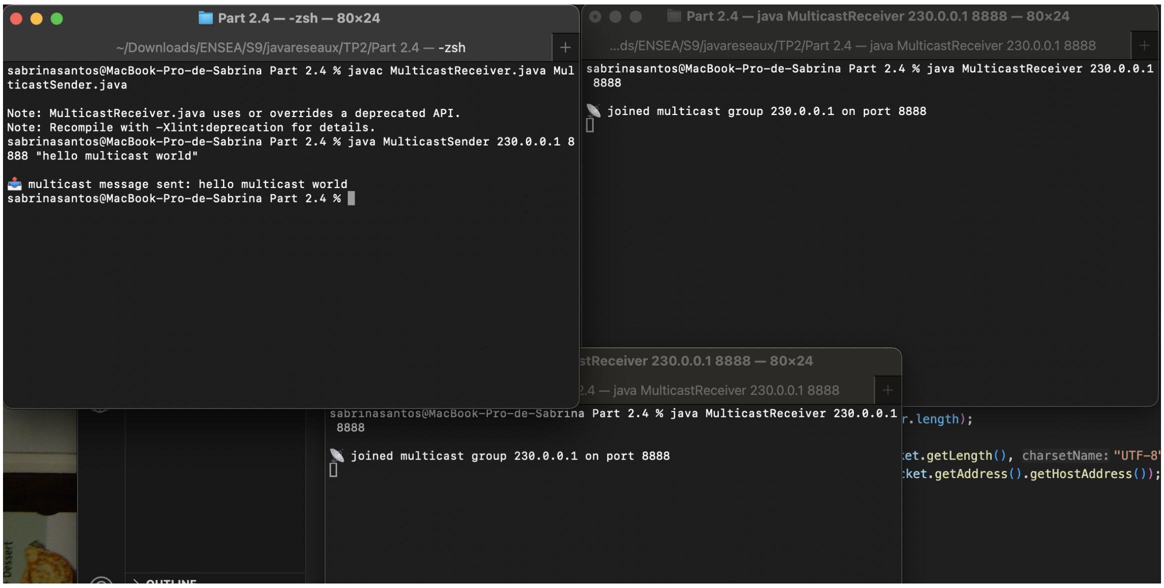


Figure 10: MacOS test

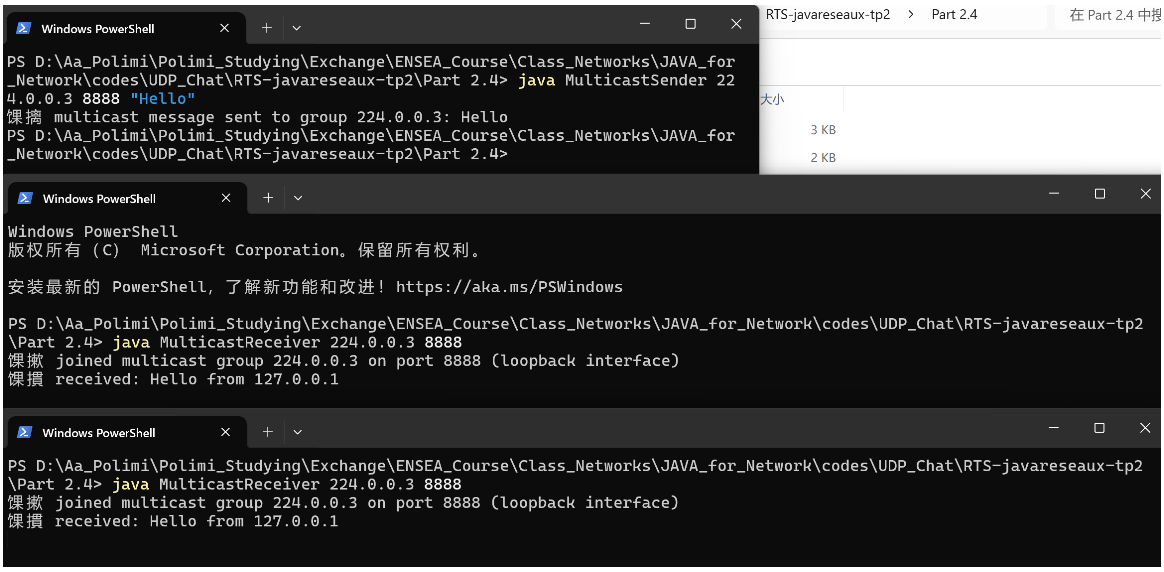


Figure 11: Windows test - receivers received