



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.

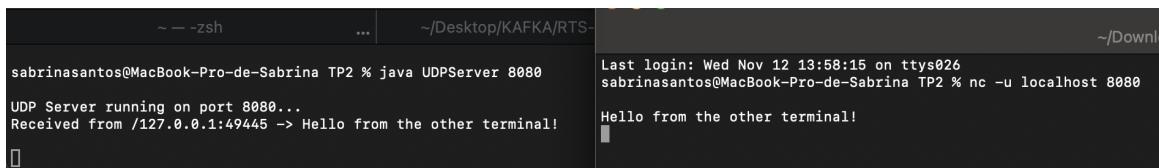


Figure 1: Testing the UDP server

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.

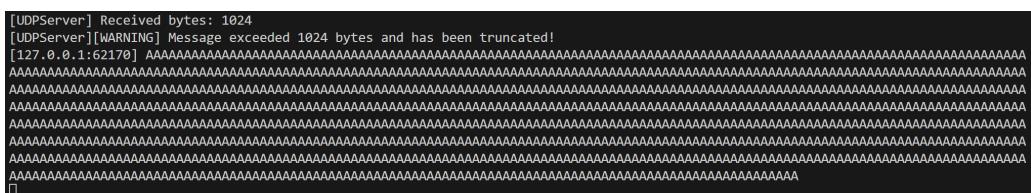


Figure 2: Testing the 1024-byte truncation on the UDP server

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 Additions

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.

```
hello!!!!
[UDPClient] Received ACK 1
testloss
[UDPClient] ACK timeout for seq 2, retry 1/3
[UDPClient] ACK timeout for seq 2, retry 2/3
[UDPClient] ACK timeout for seq 2, retry 3/3
[UDPClient][WARNING] No ACK for seq 2 after 3 retries.
Ciao, a test!
[UDPClient] Received ACK 3
^Z
[UDPClient] EOF detected. Exiting...
[UDPClient] Summary - sent=6, acked=2, lost(no-ack)=4, retransmissions=3, lossRate=66.67%
```

(a) Reliable UDP client

```
[UDPServer] Received bytes: 11
[127.0.0.1:56494] 1:hello!!!
[UDPServer] Sent ACK 1 to 127.0.0.1:56494
[UDPServer] Received bytes: 10
[127.0.0.1:56494] 2:testloss
[UDPServer] (Simulated) drop ACK for seq 2 (payload == testloss)
[UDPServer] Received bytes: 10
[127.0.0.1:56494] 2:testloss
[UDPServer] (Simulated) drop ACK for seq 2 (payload == testloss)
[UDPServer] Received bytes: 10
[127.0.0.1:56494] 2:testloss
[UDPServer] (Simulated) drop ACK for seq 2 (payload == testloss)
[UDPServer] Received bytes: 10
[127.0.0.1:56494] 2:testloss
[UDPServer] (Simulated) drop ACK for seq 2 (payload == testloss)
[UDPServer] Received bytes: 15
[127.0.0.1:56494] 3:Ciao, a test!
[UDPServer] Sent ACK 3 to 127.0.0.1:56494
```

(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

The left terminal window shows the ReliableUDPServer_2_2 application running with a small buffer size of 64 bytes. It receives a 1400 byte message from port 127.0.0.1:63099 and truncates it to 64 bytes. The right terminal window shows the ReliableUDPClient_2_2 application sending a 1400 byte payload to the server.

```

Part 2.2 — java ReliableUDPServer_2_2 8080 small — 80x24
.../ENSEA/S9/javareseaux/TP2/Part 2.2 — java ReliableUDPServer_2_2 8080 small
Last login: Wed Nov 12 16:14:12 on ttys000
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % touch ReliableUDPServer_2_2.java
|sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % touch ReliableUDPClient_2_2.java
|sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % javac ReliableUDPServer_2_2.java
ReliableUDPClient_2_2.java

sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % java ReliableUDPServer_2_2 8080
small

udp server on port 8080 using small buffer (64 bytes)

⌚ from 127.0.0.1:63099
-> bytes received: 64
-> buffer capacity: 64
-> payload (first 68 bytes): AAAAAAAAAAAAAAAA
^ truncation likely: message may be larger than buffer

Part 2.2 — zsh — 80x24
~/Downloads/ENSEA/S9/javareseaux/TP2/Part 2.2 — -zsh
Last login: Wed Nov 12 16:14:41 on ttys000
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % java ReliableUDPClient_2_2 local
host 8080 1400
sending udp datagram: payload=1400 bytes
✓ sent
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 %

```

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

The left terminal window shows the ReliableUDPServer_2_2 application running with a large buffer size of 2048 bytes. It receives a 1400 byte message from port 127.0.0.1:62994 and successfully receives the full message. The right terminal window shows the ReliableUDPClient_2_2 application sending a 1400 byte payload to the server.

```

Part 2.2 — java ReliableUDPServer_2_2 8080 large — 80x24
.../ENSEA/S9/javareseaux/TP2/Part 2.2 — java ReliableUDPServer_2_2 8080 large
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % java ReliableUDPServer_2_2 8080
large

udp server on port 8080 using large buffer (2048 bytes)

⌚ from 127.0.0.1:62994
-> bytes received: 1400
-> buffer capacity: 2048
-> payload (first 68 bytes): AAAAAAAAAAAAAAAA
✓ no truncation observed

Part 2.2 — zsh — 80x24
~/Downloads/ENSEA/S9/javareseaux/TP2/Part 2.2 — -zsh
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % java ReliableUDPClient_2_2 local
host 8080 1400
sending udp datagram: payload=1400 bytes
✓ sent
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 %

```

Figure 6: 2048 byte buffer receiving the full message

The left terminal window shows the ReliableUDPServer_2_2 application running with a small buffer size of 64 bytes. It receives a 70000 byte message from port 127.0.0.1:62994 and throws a java.net.SocketException: Message too long. The right terminal window shows the ReliableUDPClient_2_2 application sending a 70000 byte payload to the server.

```

Part 2.2 — java ReliableUDPServer_2_2 8080 small — 80x24
.../ENSEA/S9/javareseaux/TP2/Part 2.2 — java ReliableUDPServer_2_2 8080 small
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % java ReliableUDPServer_2_2 8080
small

udp server on port 8080 using small buffer (64 bytes)

⌚ from 127.0.0.1:62994
at java.base/sun.nio.ch.DatagramChannelImpl.blockingSend(DatagramChannel
Impl.java:959)
at java.base/sun.nio.ch DatagramSocketAdaptor.send(DatagramSocketAdaptor
.java:193)
at java.base/java.net.DatagramSocket.send(DatagramSocket.java:667)
at ReliableUDPClient_2_2.main(ReliableUDPClient_2_2.java:35)
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % java ReliableUDPClient_2_2 local
host 8080 70000
sending udp datagram: payload=70000 bytes
java.net.SocketException: Message too long
at java.base/sun.nio.ch.DatagramChannelImpl.send0(Native Method)
at java.base/sun.nio.ch DatagramChannelImpl.sendFromNativeBuffer(Datagra
mChannelImpl.java:1016)
at java.base/sun.nio.ch DatagramChannelImpl.send(DatagramChannelImpl.java:973)

Part 2.2 — zsh — 80x24
~/Downloads/ENSEA/S9/javareseaux/TP2/Part 2.2 — -zsh
at java.base/sun.nio.ch.DatagramChannelImpl.blockingSend(DatagramChannel
Impl.java:959)
at java.base/sun.nio.ch DatagramSocketAdaptor.send(DatagramSocketAdaptor
.java:193)
at java.base/java.net.DatagramSocket.send(DatagramSocket.java:667)
at ReliableUDPClient_2_2.main(ReliableUDPClient_2_2.java:35)
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.2 % java ReliableUDPClient_2_2 local
host 8080 70000
sending udp datagram: payload=70000 bytes
java.net.SocketException: Message too long
at java.base/sun.nio.ch.DatagramChannelImpl.send0(Native Method)
at java.base/sun.nio.ch DatagramChannelImpl.sendFromNativeBuffer(Datagra
mChannelImpl.java:1016)
at java.base/sun.nio.ch DatagramChannelImpl.send(DatagramChannelImpl.java:973)

```

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.

The screenshot shows two terminal windows. The left window, titled 'Part 2.3 - -zsh - 80x24', contains the command '/Downloads/ENSEA/S9/javareseaux/TP2/Part 2.3 - -zsh' and the output: 'sabrinasantos@MacBook-Pro-de-Sabrina Part 2.3 % java UDPClient_2_3 localhost 8080 0 "early message"'. The right window, titled 'Part 2.3 - java UDPServer_2_3 8080 - 80x24', contains the command '/Downloads/ENSEA/S9/javareseaux/TP2/Part 2.3 - java UDPServer_2_3 8080' and the output: 'sabrinasantos@MacBook-Pro-de-Sabrina Part 2.3 % java UDPServer_2_3 8080' followed by 'UDP server running on port 8080 (connectionless mode)'.

Figure 8: Client sending a message before the server starts

The screenshot shows three terminal windows. The top-left window shows a client sending a message: 'sabrinasantos@MacBook-Pro-de-Sabrina Part 2.3 % java UDPClient_2_3 localhost 8080 0 "early message"'. The top-right window shows the server receiving the message: 'sabrinasantos@MacBook-Pro-de-Sabrina Part 2.3 % java UDPServer_2_3 8080' followed by 'UDP server running on port 8080 (connectionless mode)' and 'received from 127.0.0.1:56435 -> from client 1'. The bottom-left window shows another client sending a message: 'sabrinasantos@MacBook-Pro-de-Sabrina Part 2.3 % java UDPClient_2_3 localhost 8080 0 "from client 1"' followed by '0 "from client 2"'. The bottom-right window shows the server receiving both messages: 'received from 127.0.0.1:57264 -> from client 2' and 'received from 127.0.0.1:50890 -> from client 3'.

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).

```

Part 2.4 — zsh — 80x24
~/Downloads/ENSEA/S9/javareseaux/TP2/Part 2.4 — zsh
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.4 % javac MulticastReceiver.java Mul
ticastSender.java
Note: MulticastReceiver.java uses or overrides a deprecated API.
      Recompile with -Xlint:deprecation for details.
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.4 % java MulticastSender 230.0.0.1 8
888 "Hello multicast world"
multicast message sent: Hello multicast world
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.4 %

Part 2.4 — java MulticastReceiver 230.0.0.1 8888 — 80x24
...ds/ENSEA/S9/javareseaux/TP2/Part 2.4 — java MulticastReceiver 230.0.0.1 8888
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.4 % java MulticastReceiver 230.0.0.1 8
888
joined multicast group 230.0.0.1 on port 8888

Part 2.4 — java MulticastReceiver 230.0.0.1 8888 — 80x24
...ds/ENSEA/S9/javareseaux/TP2/Part 2.4 — java MulticastReceiver 230.0.0.1 8888
sabrinasantos@MacBook-Pro-de-Sabrina Part 2.4 % java MulticastReceiver 230.0.0.1 8
888
joined multicast group 230.0.0.1 on port 8888

```

Figure 10: MacOS test

```

Windows PowerShell
PS D:\Aa_Polimi\Polimi_Studying\Exchange\ENSEA_Course\Class_Networks\JAVA_for_Network\codes\UDP_Chat\RTS-javareseaux-tp2\Part 2.4> java MulticastSender 224.0.0.3 8888 "Hello"
Hello multicast message sent to group 224.0.0.3: Hello
PS D:\Aa_Polimi\Polimi_Studying\Exchange\ENSEA_Course\Class_Networks\JAVA_for_Network\codes\UDP_Chat\RTS-javareseaux-tp2\Part 2.4>

Windows PowerShell
版权所有 (C) Microsoft Corporation. 保留所有权利。
安装最新的 PowerShell, 了解新功能和改进! https://aka.ms/PSWindows

PS D:\Aa_Polimi\Polimi_Studying\Exchange\ENSEA_Course\Class_Networks\JAVA_for_Network\codes\UDP_Chat\RTS-javareseaux-tp2\Part 2.4> java MulticastReceiver 224.0.0.3 8888
joined multicast group 224.0.0.3 on port 8888 (loopback interface)
received: Hello from 127.0.0.1

Windows PowerShell
PS D:\Aa_Polimi\Polimi_Studying\Exchange\ENSEA_Course\Class_Networks\JAVA_for_Network\codes\UDP_Chat\RTS-javareseaux-tp2\Part 2.4> java MulticastReceiver 224.0.0.3 8888
joined multicast group 224.0.0.3 on port 8888 (loopback interface)
received: Hello from 127.0.0.1

```

Figure 11: Windows test - receivers received