David LE, Mehdi BOULHARTS, 3A RTS
Supervisor: Linda SENIGAGLIESI

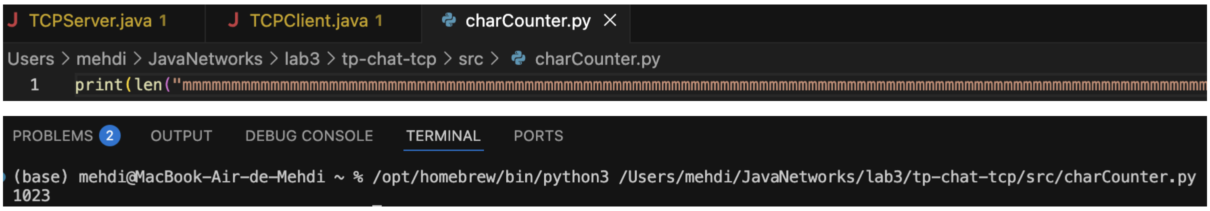
TP3 – Java for networks

3.7 Testing and Protocol Comparison

3.7.1 Basic functionality test – long message and special characters

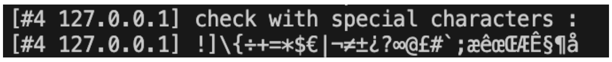
Long message

server :



Special characters

server :



To test the basic behaviour of our TCP chat, we first sent a very long line generated by a Python script. The server correctly echoed the message up to 1023 characters. Sending 1024 characters failed, which suggests a practical line-length limit in our client/terminal setup.
We then sent a line containing many special characters (punctuation and accented letters). The server displayed and echoed the full string without corruption, showing that our UTF-8 based implementation can handle long messages and extended characters correctly.

3.7.3 Multiple client test

server :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPServer 9000
TCPServer(port=9000)
[16:26:40] Server started on port 9000
[16:26:44] New connection from 127.0.0.1 (client #1)
[#1 127.0.0.1] hello
[16:26:49] New connection from 127.0.0.1 (client #2)
[#2 127.0.0.1] hi
[16:27:06] New connection from 127.0.0.1 (client #3)
[#3 127.0.0.1] hola
[#1 127.0.0.1] test client1
[#3 127.0.0.1] test client3
[#2 127.0.0.1] test client2
[#2 127.0.0.1] again cleint2
[#3 127.0.0.1] and now client3
[#1 127.0.0.1] in the end client1
[16:28:31] New connection from 127.0.0.1 (client #4)
[#4 127.0.0.1] it works
```

clients :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPClient localhost 9000
Connecting to localhost:9000 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
> hello
[#1 127.0.0.1] hello
> test client1
[#1 127.0.0.1] test client1
> in the end client1
[#1 127.0.0.1] in the end client1
```

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPClient localhost 9000
Connecting to localhost:9000 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
> hi
[#2 127.0.0.1] hi
> test client2
[#2 127.0.0.1] test client2
> again cleint2
[#2 127.0.0.1] again cleint2
```

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPClient localhost 9000
Connecting to localhost:9000 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
> hola
[#3 127.0.0.1] hola
> test client3
[#3 127.0.0.1] test client3
> and now client3
[#3 127.0.0.1] and now client3
```

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPClient localhost 9000
Connecting to localhost:9000 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
> it works
[#4 127.0.0.1] it works
```

In this experiment we started the TCPServer on port 9000 and connected four different TCPClient instances. Each new connection was logged on the server with a timestamp and a unique client ID.
Every client could send messages independently and received its own echo prefixed with [#id ip]. The server handled all clients concurrently without blocking or errors, which confirms that the threaded design supports multiple simultaneous connections.

3.7.4 Error scenario testing

Abrupt disconnection

close properly with /quit client 3

```
[15:19:09] Client #3 (127.0.0.1) disconnected
```

close the window of my shell for client 4

```
[15:18:27] Client #4 (127.0.0.1) disconnected
```

There is no difference between them so no issue here. We tried with another strategy to make an abrupt disconnection :

shell to kill the client :

```
mehdi           22482   0,0  0,3 414658320  27952 s006  S+    4:05     0:00.11 /usr/bin/java -cp out TCPClient localhost 9000
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % kill -9 22482
```

server :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPServer 9000
TCPServer(port=9000)
[16:05:28] Server started on port 9000
[16:05:31] New connection from 127.0.0.1 (client #1)
[16:07:38] Client #1 (127.0.0.1) disconnected
```

client :

```
> zsh: killed     java -cp out TCPClient localhost 9000
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp %
```

And then we see that it's again detected as a normal disconnection.

In our implementation, both graceful and abrupt client disconnections are handled correctly. When the client closes normally, the server's readLine() call returns null and the handler terminates the connection gracefully.
In the case of an abrupt disconnection (ex : killing the client process), an IOException is thrown and caught, the error is logged, and the server still closes the socket in the finally block.
From the server's point of view, both scenarios result in a clean resource cleanup.

Client connection to non-existent server

server :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPServer 9000
TCPServer(port=9000)
[15:25:08] Server started on port 9000
[15:25:08] New connection from 127.0.0.1 (client #1)
[#1 127.0.0.1] I tried twice to connect myself to tcp server with no success. One more and I'd have stopped !
```

client :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPClient localhost 9000
Connecting to localhost:9000 (attempt 1/3)...
Connection failed: Connection refused
Reconnecting in 2 seconds...
Connecting to localhost:9000 (attempt 2/3)...
Connection failed: Connection refused
Reconnecting in 2 seconds...
Connecting to localhost:9000 (attempt 3/3)...
Connection failed: Connection refused
Maximum number of attempts reached. Giving up.
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPClient localhost 9000
Connecting to localhost:9000 (attempt 1/3)...
Connection failed: Connection refused
Reconnecting in 2 seconds...
Connecting to localhost:9000 (attempt 2/3)...
Connection failed: Connection refused
Reconnecting in 2 seconds...
Connecting to localhost:9000 (attempt 3/3)...
Connection established.
Type messages to send. Use /quit to exit.
> I tried twice to connect myself to tcp server with no success. One more and I'd have stopped !
[#1 127.0.0.1] I tried twice to connect myself to tcp server with no success. One more and I'd have stopped !
```

We tried to connect our client to our server but this server was not already launched. We can see that when we launch the server quickly enough, our client achieves the connection. But if it takes too much time, the client gives up (he tries each 2 seconds 3 times).

Server disconnection while a client is connected

server :

```
[#1 127.0.0.1] I tried twice to connect myself to tcp server with no success. One more and I'd have stopped !
^C%
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp %
```

client :

```
[#1 127.0.0.1] I tried twice to connect myself to tcp server with no success. One more and I'd have stopped !
> hello
Connection failed: Connection reset
Maximum number of attempts reached. Giving up.
```

We shutted down the server and we lost the connection only when we tried to send the message from the client. The client tries 3 times maximum to connect to the server so he gives up. Here another example with a client launched after the server :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPClient localhost 9000
Connecting to localhost:9000 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
>
> m
Connection failed: Connection reset
Reconnecting in 2 seconds...
Connecting to localhost:9000 (attempt 2/3)...
Connection failed: Connection refused
Reconnecting in 2 seconds...
Connecting to localhost:9000 (attempt 3/3)...
Connection failed: Connection refused
Maximum number of attempts reached. Giving up.
```

Conclusion

Through these tests, we verified that our TCP chat implementation behaves correctly in normal and abnormal situations. The server accepts multiple simultaneous clients, handles long messages and special characters, and cleans up connections properly. On the client side, the reconnection mechanism prevents the application from hanging when the server is unavailable or crashes. Overall, the system is robust enough for interactive use and provides a good practical illustration of TCP's connection-oriented and reliable nature.

3.8 UDP vs TCP Comparison

Testing Different TCP and UDP Communication Modes

Two independent UDP datagrams sent from the client to the server.

1	0.000000	127.0.0.1	127.0.0.1	UDP	37	58326 → 8080 Len=5
2	14.821857	127.0.0.1	127.0.0.1	UDP	47	58326 → 8080 Len=15

There is no connection setup and no extra control packets: each send() directly produces one UDP packet from source port 58326 to destination port 8080, carrying the application payload (Len=5 and Len=15).

Capture of the ReliableUDP exchange with 50 messages and a 0% drop rate.

3	70.123104	127.0.0.1	127.0.0.1	UDP	46	55840 → 8080 Len=14
4	70.124199	127.0.0.1	127.0.0.1	UDP	37	8080 → 55840 Len=5
5	70.124379	127.0.0.1	127.0.0.1	UDP	46	55840 → 8080 Len=14
6	70.124607	127.0.0.1	127.0.0.1	UDP	37	8080 → 55840 Len=5
100	70.134902	127.0.0.1	127.0.0.1	UDP	37	8080 → 55840 Len=5
101	70.134956	127.0.0.1	127.0.0.1	UDP	47	55840 → 8080 Len=15
102	70.135077	127.0.0.1	127.0.0.1	UDP	37	8080 → 55840 Len=5

For each application message sent by the client from port 55840 to server port 8080, we see a corresponding ACK packet in the opposite direction.
In total, 50 messages generate almost twice as many UDP packets (data + acknowledgements), but all messages are delivered successfully.

ReliableUDP with a simulated 30% drop rate.

121	324.788003	127.0.0.1	127.0.0.1	UDP	45	59726 → 8080 Len=13
122	324.789052	127.0.0.1	127.0.0.1	UDP	37	8080 → 59726 Len=5
123	324.789240	127.0.0.1	127.0.0.1	UDP	45	59726 → 8080 Len=13
124	324.991312	127.0.0.1	127.0.0.1	UDP	45	59726 → 8080 Len=13
236	328.261356	127.0.0.1	127.0.0.1	UDP	47	59726 → 8080 Len=15
237	328.462621	127.0.0.1	127.0.0.1	UDP	47	59726 → 8080 Len=15
238	328.463173	127.0.0.1	127.0.0.1	UDP	37	8080 → 59726 Len=5

Some data packets sent from the client (port 59726 → 8080) are not acknowledged and must be retransmitted.
Wireshark shows repeated packets with the same sequence number and additional ACKs.
The client can still deliver most messages reliably, but at the cost of extra traffic and higher latency.

MulticastUDP on group 230.0.0.1:8888.

1	0.000000	172.20.10.2	230.0.0.1	UDP	47	57043 → 8888 Len=15

A single datagram is sent from the sender (172.20.10.2, port 57043) to the multicast group address 230.0.0.1:8888 (Len=15).
Both receivers that joined this group obtain a copy of the same packet, which illustrates one-to-many delivery with a single send operation.

TCP connection on localhost between ports 53774 and 9000.

2	51.523126	127.0.0.1	127.0.0.1	TCP	68	53774 → 9000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=4020618663 TSecr=0 SACK_PERM
3	51.523254	127.0.0.1	127.0.0.1	TCP	68	9000 → 53774 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=2705167114 TSecr=4020618663
4	51.523279	127.0.0.1	127.0.0.1	TCP	56	53774 → 9000 [ACK] Seq=1 Ack=1 Win=408320 Len=0 TSval=4020618663 TSecr=2705167114
5	51.523296	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 9000 → 53774 [ACK] Seq=1 Ack=1 Win=408320 Len=0 TSval=2705167114 TSecr=4020618663
6	55.884039	127.0.0.1	127.0.0.1	TCP	62	53774 → 9000 [PSH, ACK] Seq=1 Ack=1 Win=408320 Len=6 TSval=4020623024 TSecr=2705167114
7	55.884087	127.0.0.1	127.0.0.1	TCP	56	9000 → 53774 [ACK] Seq=1 Ack=7 Win=408320 Len=0 TSval=2705171475 TSecr=4020623024
8	55.897602	127.0.0.1	127.0.0.1	TCP	77	9000 → 53774 [PSH, ACK] Seq=1 Ack=7 Win=408320 Len=21 TSval=2705171489 TSecr=4020623024
9	55.897668	127.0.0.1	127.0.0.1	TCP	56	53774 → 9000 [ACK] Seq=7 Ack=22 Win=408320 Len=0 TSval=4020623038 TSecr=2705171489
10	60.880301	127.0.0.1	127.0.0.1	TCP	72	53774 → 9000 [PSH, ACK] Seq=7 Ack=22 Win=408320 Len=16 TSval=4020628021 TSecr=2705171489
11	60.880380	127.0.0.1	127.0.0.1	TCP	56	9000 → 53774 [ACK] Seq=22 Ack=23 Win=408320 Len=0 TSval=2705176472 TSecr=4020628021
12	60.881105	127.0.0.1	127.0.0.1	TCP	87	9000 → 53774 [PSH, ACK] Seq=22 Ack=23 Win=408320 Len=31 TSval=2705176472 TSecr=4020628021
13	60.881135	127.0.0.1	127.0.0.1	TCP	56	53774 → 9000 [ACK] Seq=23 Ack=53 Win=408320 Len=0 TSval=4020628021 TSecr=2705176472
14	64.225659	127.0.0.1	127.0.0.1	TCP	56	53774 → 9000 [FIN, ACK] Seq=23 Ack=53 Win=408320 Len=0 TSval=4020631366 TSecr=2705176472
15	64.225730	127.0.0.1	127.0.0.1	TCP	56	9000 → 53774 [ACK] Seq=53 Ack=24 Win=408320 Len=0 TSval=2705179817 TSecr=4020631366
16	64.228328	127.0.0.1	127.0.0.1	TCP	56	9000 → 53774 [FIN, ACK] Seq=53 Ack=24 Win=408320 Len=0 TSval=2705179820 TSecr=4020631366
17	64.228420	127.0.0.1	127.0.0.1	TCP	56	53774 → 9000 [ACK] Seq=24 Ack=54 Win=408320 Len=0 TSval=4020631369 TSecr=2705179820

At the beginning we observe the three-way handshake: SYN, SYN, ACK, then ACK.
This is followed by two PSH, ACK segments carrying the application data.
Finally, both endpoints close the connection with a FIN, ACK exchange in each direction.
Compared to UDP, we see additional control packets for connection establishment and termination, even though only two application messages are transmitted.

3.8.1 Performance comparison

From the UDP traces, we see that there is no connection setup: the client can send its first datagram immediately and only the application packets appear on the wire. In contrast, the TCP trace shows a three-way handshake before the two data segments and a FIN/ACK exchange to close the session, which adds control overhead and slightly increases the connection establishment time.

Regarding reliability, basic UDP and multicast do not include acknowledgements: if a packet is lost, it simply disappears. With ReliableUDP we had to add sequence numbers, ACKs and retransmissions, which increases the number of packets and the latency, especially with a 30% drop rate. TCP integrates these mechanisms directly in the protocol, providing reliable and ordered delivery without extra logic in the application, at the cost of additional protocol overhead.

3.8.2 Behavioral differences

With UDP, communication is message-oriented: each send() call in the client produces exactly one datagram, and the server receives the same message boundaries. This fits small independent messages and multicast, where one datagram can be delivered to several receivers. However, the protocol is connectionless and best-effort: there is no built-in feedback, so any reliability or timeout logic (as in ReliableUDP) must be implemented in the application.

TCP is stream-oriented and connection-oriented. The application sees a continuous byte stream, and message boundaries are reconstructed by higher-level code (for example using readLine). Both endpoints maintain connection state and the protocol handles acknowledgements, retransmissions, flow control and congestion control internally. As a result, the application mostly deals with opening/closing connections and reading/writing data, not with individual packet losses.

3.9 Additional Enhancement Activities

Tests for the enhanced protocol and session management

sever :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out NewTCPServer 9090
TCPServer(port=9090)
[11:13:54] Server started on port 9090
[11:13:58] New connection from 127.0.0.1 (client #1)
[11:14:00] Protocol error with client #1 (127.0.0.1): Invalid header (expected 4
 fields): hello
[11:14:48] Session for client #1 (127.0.0.1) closed. Duration = 50 s
[11:14:48] Client #1 (127.0.0.1) disconnected
[11:14:57] New connection from 127.0.0.1 (client #2)
[#2 127.0.0.1 seq=1 type=CHAT len=5] hello
[11:16:00] Session timeout for client #2 (127.0.0.1) after 60000 ms of inactivit
y
[11:16:00] Session for client #2 (127.0.0.1) closed. Duration = 63 s
[11:16:00] Client #2 (127.0.0.1) disconnected
```

client1:

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out TCPClient localhost
 9090
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
> hello
ERROR|0|57|Protocol error: Invalid header (expected 4 fields): hello
```

client2 :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp % java -cp out NewTCPClient localh
ost 9090
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
> hello
[server seq=1 type=SYSTEM len=5] hello
```

In the first test (client1), we use the old text-only TCP client with the new server. The client sends the string hello without any header, while the server now expects the format TYPE|SEQ|LEN|PAYLOAD. The server logs a “Protocol error: Invalid header (expected 4 fields)” and automatically replies with an ERROR message, which is displayed on the client side. This demonstrates the new application-level protocol and the advanced error handling using ProtocolException.
In the second test (client2), we use the new TCP client that correctly formats messages as CHAT|seq|len|payload. The server logs the decoded fields (seq=1 type=CHAT len=5) and answers with a SYSTEM message echoing the payload. After that, we stop sending data: after 60 seconds of inactivity, the server triggers the session timeout, logs “Session timeout … after 60000 ms of inactivity” and closes the connection. This validates the session management with per-client IDs, session duration logging, and automatic cleanup of idle connections.

3.9.1 Message formatting protocol

In the enhanced version of the chat, each message is encoded with a small application-level header: TYPE|SEQ|LEN|PAYLOAD.
TYPE indicates whether it is a normal chat message, a system response or an error; SEQ is a sequence number, and LEN is the payload length. The server decodes this header using the ChatMessage class and logs fields such as seq, type and len for each client message.

3.9.2 Session management

The server assigns a unique client ID to every new TCP connection and treats it as a session. For each session it records the start time and prints the total duration when the client disconnects. We also configured a 60-second inactivity timeout using setSoTimeout(): if no message is received during this period, the server logs a “Session timeout” event and closes the socket. This ensures that idle connections are cleaned up automatically.

3.9.3 Advanced error handling

When the message format is invalid (for example, when using the old text-only client with the new server), the ChatMessage.decode() method throws a ProtocolException. The server catches this exception, logs the error and sends an ERROR message back to the client. On the client side, error messages are displayed with a clear prefix ([SERVER ERROR]or [PROTOCOL ERROR]). Network errors such as connection refusal or timeouts are still handled with retries as in the previous version, so the application does not hang when the server is unavailable or crashes.