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

TP4 – Java for networks

4 Step-by-Step Exercises

4.1 Exercise 1: Basic Multithreading Implementation

We tried to verify that the multithreaded TCP server works correctly with a single client: the server must accept the connection, send a welcome message, echo back each line, and allow a clean disconnection with a quit command.

We implemented ConnectionThread (one thread per client) and MultithreadedTCPServer, then started the server on port 9090. From another terminal, we launched TCPClient localhost 9090, sent a test message (hello), and then used the quit command to close the session.

server :

```
MultithreadedTCPServer(port=9090)
Multithreaded Server started on port 9090
=== Thread Statistics ===
 Active threads : 1
 Memory usage : 3147 KB
[Wed Nov 26 15:21:46 CET 2025] Client 1 connected from 127.0.0.1
[#1 127.0.0.1] hello
[Wed Nov 26 15:25:51 CET 2025] Client 1 (127.0.0.1) disconnected
```

client :

```
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #1
> hello
[#1 127.0.0.1] hello
> quit
User requested to quit. Closing connection.
Disconnected from server. Bye.
```

The client successfully connected, printed Welcome! You are client #1, and received the echoed message for hello. When typing quit, the client received the goodbye message and disconnected cleanly, while the server logged the end of the session. This confirms that the basic behaviour required is correctly implemented.

4.2 Exercise 2: Concurrent Client Testing

We wanted to test the multithreaded TCP server with several clients connected at the same time, to check that each client is handled by its own thread and that their communications remain independent.

We started the MultithreadedTCPServer on the lab port, then opened three separate terminals and launched TCPClient localhost <port> in each of them. Each client sent a few messages and then used the quit command to disconnect.

server :

```
MultithreadedTCPServer(port=9090)
Multithreaded Server started on port 9090
=== Thread Statistics ===
 Active threads : 1
 Memory usage : 3147 KB
[Wed Nov 26 15:36:12 CET 2025] Client 1 connected from 127.0.0.1
=== Thread Statistics ===
 Active threads : 2
 Memory usage : 4175 KB
[Wed Nov 26 15:36:15 CET 2025] Client 2 connected from 127.0.0.1
=== Thread Statistics ===
 Active threads : 3
 Memory usage : 4212 KB
[Wed Nov 26 15:36:17 CET 2025] Client 3 connected from 127.0.0.1
[#1 127.0.0.1] hello1
[#2 127.0.0.1] hello2
[#3 127.0.0.1] hello3
[#2 127.0.0.1] first test client2
[#1 127.0.0.1] another test client1
[#3 127.0.0.1] last test client3
[Wed Nov 26 15:37:58 CET 2025] Client 3 (127.0.0.1) disconnected
[Wed Nov 26 15:38:01 CET 2025] Client 2 (127.0.0.1) disconnected
[Wed Nov 26 15:38:04 CET 2025] Client 1 (127.0.0.1) disconnected
```

clients :

```
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #1
> hello1
[#1 127.0.0.1] hello1
> another test client1
[#1 127.0.0.1] another test client1
> quit
User requested to quit. Closing connection.
Disconnected from server. Bye.
```

```
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #2
> hello2
[#2 127.0.0.1] hello2
> first test client2
[#2 127.0.0.1] first test client2
> quit
User requested to quit. Closing connection.
Disconnected from server. Bye.
```

```
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #3
> hello3
[#3 127.0.0.1] hello3
> last test client3
[#3 127.0.0.1] last test client3
> quit
User requested to quit. Closing connection.
Disconnected from server. Bye.
```

The three clients connected successfully with sequential IDs (#1, #2, #3). Each client received only its own echoed messages, correctly tagged with its client ID, while the server console showed interleaved messages from all clients and an increasing active thread count. This confirms that concurrent client handling works as expected.

4.3 Exercise 3: Thread Safety Validation

Our objective was to validate the thread safety of the multithreaded server by starting many clients nearly at the same time and checking that the shared client counter produces unique IDs without race conditions.

We kept MultithreadedTCPServer running on port 9090 and executed a shell loop that launched ten TCPClient localhost 9090 instances in parallel. Each client automatically sent the message hello from client i, then /quit to close the connection.

server :

```
Multithreaded Server started on port 9090
=== Thread Statistics ===
 Active threads : 1
 Memory usage : 2655 KB
=== Thread Statistics ===
 Active threads : 2
 Memory usage : 3147 KB
=== Thread Statistics ===
 Active threads : 3
 Memory usage : 3188 KB
=== Thread Statistics ===
 Active threads : 4
 Memory usage : 3679 KB
=== Thread Statistics ===
 Active threads : 5
 Memory usage : 4171 KB
=== Thread Statistics ===
 Active threads : 6
 Memory usage : 4703 KB
=== Thread Statistics ===
 Active threads : 7
 Memory usage : 6260 KB
=== Thread Statistics ===
 Active threads : 8
 Memory usage : 6751 KB
[Wed Nov 26 16:00:31 CET 2025] Client 8 connected from 127.0.0.1
[Wed Nov 26 16:00:31 CET 2025] Client 5 connected from 127.0.0.1
[Wed Nov 26 16:00:31 CET 2025] Client 7 connected from 127.0.0.1
[Wed Nov 26 16:00:31 CET 2025] Client 6 connected from 127.0.0.1
[Wed Nov 26 16:00:31 CET 2025] Client 2 connected from 127.0.0.1
[Wed Nov 26 16:00:31 CET 2025] Client 1 connected from 127.0.0.1
[Wed Nov 26 16:00:31 CET 2025] Client 3 connected from 127.0.0.1
[Wed Nov 26 16:00:31 CET 2025] Client 4 connected from 127.0.0.1
[#3 127.0.0.1] hello from client 8
[#2 127.0.0.1] hello from client 3
[#7 127.0.0.1] hello from client 9
[Wed Nov 26 16:00:31 CET 2025] Client 2 (127.0.0.1) disconnected
[Wed Nov 26 16:00:31 CET 2025] Client 7 (127.0.0.1) disconnected
[#8 127.0.0.1] hello from client 5
[#1 127.0.0.1] hello from client 2
[Wed Nov 26 16:00:31 CET 2025] Client 3 (127.0.0.1) disconnected
[Wed Nov 26 16:00:31 CET 2025] Client 8 (127.0.0.1) disconnected
[Wed Nov 26 16:00:31 CET 2025] Client 1 (127.0.0.1) disconnected
[#5 127.0.0.1] hello from client 6
[#6 127.0.0.1] hello from client 10
[#4 127.0.0.1] hello from client 1
[Wed Nov 26 16:00:31 CET 2025] Client 6 (127.0.0.1) disconnected
[Wed Nov 26 16:00:31 CET 2025] Client 4 (127.0.0.1) disconnected
[Wed Nov 26 16:00:31 CET 2025] Client 5 (127.0.0.1) disconnected
=== Thread Statistics ===
 Active threads : 1
 Memory usage : 7243 KB
[Wed Nov 26 16:00:31 CET 2025] Client 9 connected from 127.0.0.1
[#9 127.0.0.1] hello from client 4
[Wed Nov 26 16:00:31 CET 2025] Client 9 (127.0.0.1) disconnected
=== Thread Statistics ===
 Active threads : 1
 Memory usage : 7775 KB
[Wed Nov 26 16:00:31 CET 2025] Client 10 connected from 127.0.0.1
[#10 127.0.0.1] hello from client 7
[Wed Nov 26 16:00:31 CET 2025] Client 10 (127.0.0.1) disconnected
```

clients :

```
Type messages to send. Use /quit to exit.
Connection established.
Type messages to send. Use /quit to exit.
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #7
> 9090 (attempt 1/3)...
Welcome! You are client #3
> Welcome! You are client #2
> [#2 127.0.0.1] hello from client 3
> User requested to quit. Closing connection.
[#7 127.0.0.1] hello from client 9
> User requested to quit. Closing connection.
[#3 127.0.0.1] hello from client 8
Disconnected from server. Bye.
Disconnected from server. Bye.
> User requested to quit. Closing connection.
Welcome! You are client #1
> Welcome! You are client #8
> Disconnected from server. Bye.
[#8 127.0.0.1] hello from client 5
> User requested to quit. Closing connection.
[#1 127.0.0.1] hello from client 2
> User requested to quit. Closing connection.
9090 (attempt 1/Disconnected from server. Bye.
3[10]  - done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
)...
[4]    done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
Welcome! You are client #5
> Welcome! You are client #4
> [9]  - done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
Disconnected from server. Bye.
Welcome! You are client #6
[#5 127.0.0.1] hello from client 6
> > User requested to quit. Closing connection.
[6]    done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
[#6 127.0.0.1] hello from client 10
> User requested to quit. Closing connection.
[#4 127.0.0.1] hello from client 1
> User requested to quit. Closing connection.
Disconnected from server. Bye.
Disconnected from server. Bye.
Disconnected from server. Bye.
[3]    done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
Connection established.
Type messages to send. Use /quit to exit.
[11]  + done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
[2]    done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
[7]  - done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
Welcome! You are client #9
> [#9 127.0.0.1] hello from client 4
> User requested to quit. Closing connection.
Disconnected from server. Bye.
[5]  - done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #10
> [#10 127.0.0.1] hello from client 7
> User requested to quit. Closing connection.
Disconnected from server. Bye.
[8]  + done       printf "hello from client $i\n/quit\n" | java -cp out TCPClient localhost 9090
```

On the server console we observed ten connections with client IDs from #1 to #10, all sending their “hello” messages and disconnecting cleanly, while the active thread count increased and then went back to 1. No errors or duplicate IDs appeared, which confirms that the AtomicInteger-based client ID generation is thread-safe in this stress test.

Conclusion

These exercises showed that the basic “one thread per client” design works correctly : clients can connect, exchange messages and disconnect without interfering with each other.

5 Advanced Implementation

5.1 Thread Pool Enhancement

Here, we wanted to replace the “one thread per client” design with a fixed-size thread pool, so that the server keeps the same behaviour for each client but limits the number of concurrent handler threads (better control of resources under high load).

We implemented ThreadPoolTCPServer, which uses an ExecutorService with a fixed pool of 10 threads. For each new connection, the server increments the AtomicInteger client counter, then submits a task that runs a ConnectionThreadfor that client. We first tested with a few manual clients (connect, send hello, /quit), then ran a stress test with 50 clients started in parallel from the shell.

server :

```
ThreadPoolTCPServer(port=9090)
Thread Pool Server started on port 9090
=== Thread Statistics ===
 Active threads : 1
 Memory usage : 3147 KB
[Wed Nov 26 16:21:16 CET 2025] Client 1 connected from 127.0.0.1
=== Thread Statistics ===
 Active threads : 2
 Memory usage : 4171 KB
[Wed Nov 26 16:21:22 CET 2025] Client 2 connected from 127.0.0.1
[#1 127.0.0.1] hello1
[Wed Nov 26 16:22:12 CET 2025] Client 1 (127.0.0.1) disconnected
[#2 127.0.0.1] hello2
[Wed Nov 26 16:22:24 CET 2025] Client 2 (127.0.0.1) disconnected
```

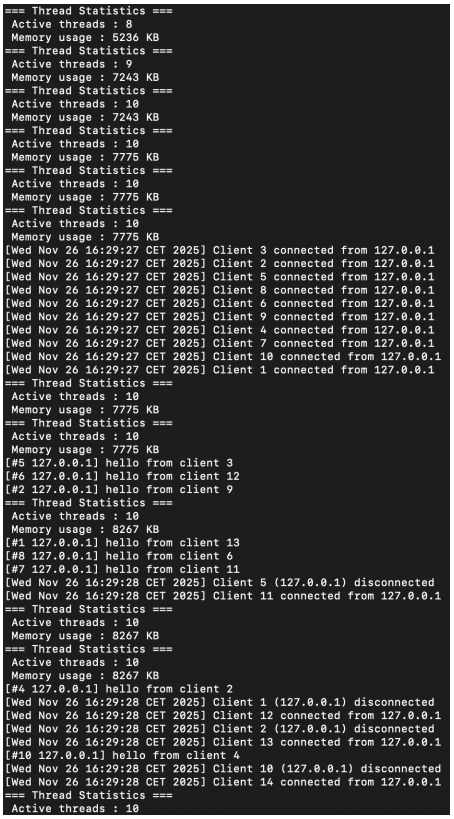
clients :

```
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #1
> hello1
[#1 127.0.0.1] hello1
> quit
User requested to quit. Closing connection.
Disconnected from server. Bye.
```

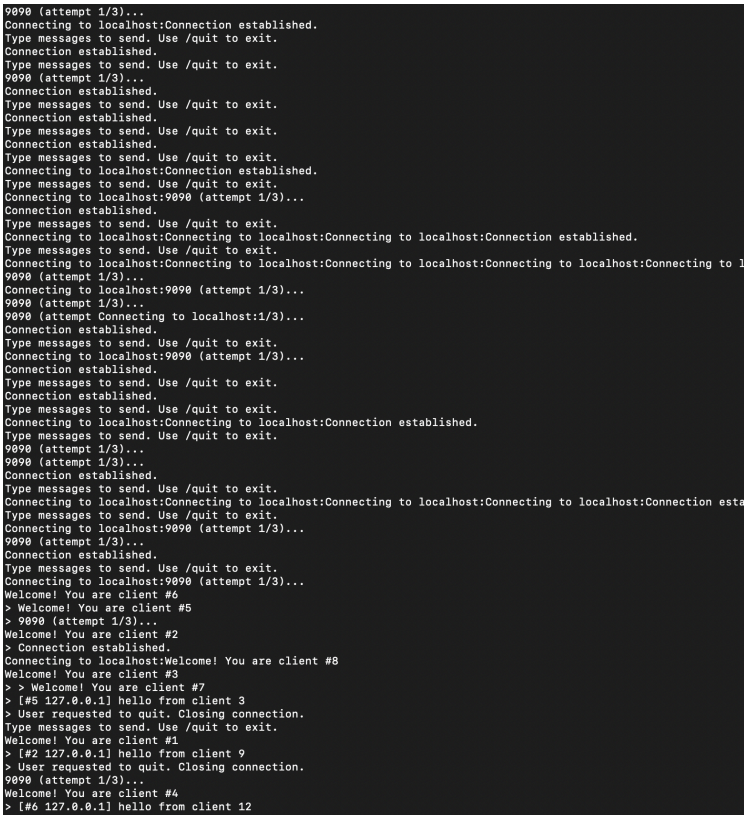
```
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #2
> hello2
[#2 127.0.0.1] hello2
> quit
User requested to quit. Closing connection.
Disconnected from server. Bye.
```

In the small test, each client still received its welcome message (client #1, client #2), got its echo, and disconnected cleanly, showing that the thread pool does not change the observable behaviour.

server :



clients :



In the 50-client stress test, the server logs showed client IDs increasing up to #50 while the “Active threads” value quickly climbed to 10 and then stayed around this limit, with all clients eventually disconnecting without errors. This confirms that the thread pool correctly bounds the number of worker threads while still handling a large number of clients.

Conclusion

With the thread pool, the server keeps the same behaviour for clients but caps the number of worker threads, which makes resource usage more predictable under higher load.

6 Testing and Validation

6.1 Comprehensive Test Plan

The objective was to validate the global behaviour and robustness of our TCP chat system, both for basic functionality (single + multiple clients) and for more realistic usage patterns such as repeated connections from the same terminal.

For functionality, we reused the scenarios from Exercises 4.1–4.3 and Section 5: single client lifecycle, several concurrent clients, stress tests with 10 and 50 clients, and the thread-pool server (see screenshots in those parts of the report). Now, we added an explicit reconnection test: we kept MultithreadedTCPServer running, connected once with TCPClient localhost 9090 (client #1, message, /quit), then re-launched the same client from the same terminal and checked that it connected again as a new client (#2), as shown in the screenshots below.

server :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp-multi % java -cp out MultithreadedTCPServer 9090
MultithreadedTCPServer(port=9090)
Multithreaded Server started on port 9090
=== Thread Statistics ===
 Active threads : 1
 Memory usage : 3147 KB
[Wed Nov 26 16:52:25 CET 2025] Client 1 connected from 127.0.0.1
[#1 127.0.0.1] hello
[Wed Nov 26 16:52:31 CET 2025] Client 1 (127.0.0.1) disconnected
=== Thread Statistics ===
 Active threads : 1
 Memory usage : 4212 KB
[Wed Nov 26 16:52:40 CET 2025] Client 2 connected from 127.0.0.1
```

clients :

```
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp-multi % java -cp out TCPClient localhost 9090
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #1
> hello
[#1 127.0.0.1] hello
> /quit
User requested to quit. Closing connection.
Disconnected from server. Bye.
(base) mehdi@MacBook-Air-de-Mehdi tp-chat-tcp-multi % java -cp out TCPClient localhost 9090
Connecting to localhost:9090 (attempt 1/3)...
Connection established.
Type messages to send. Use /quit to exit.
Welcome! You are client #2
>
```

All tests confirm that the server assigns unique client IDs, isolates messages per client, and cleans up connections correctly. The reconnection test shows that the same user can close a session and later reconnect with a new ID without any error, which completes the functional and basic robustness validation required.

6.2 Performance Monitoring

To monitor the behaviour of the server under load, we added a printThreadStats() method that prints the current number of active threads and the JVM memory usage in KB (see the server logs in Sections 4.2, 4.3 and 5). This method is called each time a new client connects, both in MultithreadedTCPServer and in ThreadPoolTCPServer.

During the earlier stress tests (10 and then 50 clients), we used these logs to track the main performance metrics: the active thread count increased with each new connection and then decreased when clients disconnected, while the memory usage grew from a few MB up to around 7–8 MB before stabilising. With the thread-per-client server, the number of active threads followed the number of concurrent clients, whereas with the thread-pool server it quickly reached the pool limit (10 threads) and then stayed around that value even when launching 50 clients.

These observations show that the server keeps a reasonable memory footprint and that the thread-pool version effectively bounds the number of worker threads while still accepting many client connections in a short time. Combined with the previous functional and robustness tests, this confirms that the multithreaded architecture behaves as expected from a performance point of view in our lab environment.

CPU utilisation was briefly checked with the system monitor and remained low with short spikes during stress tests.

Conclusion
The tests and monitoring confirm that the system stays stable and responsive in normal, concurrent and stress scenarios, so the implementation meets the reliability goals of the lab.