Java network concurrency - Course material

https://github.com/heig-vd-dai-course

Markdown · PDF

L. Delafontaine and H. Louis, with the help of GitHub Copilot.

Based on the original course by O. Liechti and J. Ehrensberger.

This work is licensed under the CC BY-SA 4.0 license.

Table of contents

- Table of contents
- Objectives
- TCP
- The Socket API
    - Client/server common methods
    - Client workflow and methods
    - Server structure and methods
- Processing data from streams
    - Variable length data
- Handling one client at a time
- Handling multiple clients with concurrency
    - Multi-processing
    - Multi-threading
    - Asynchronous programming
- Practical content
    - Get the required files
    - Send an email using a SMTP client written in Java with the Socket API
    - Run full client/server examples
    - Go further
- Conclusion
    - What did you do and learn?
    - Test your knowledge
- Finished? Was it easy? Was it hard?
- What will you do next?
- Additional resources
- Solution
- Sources

Objectives

As you have seen in previous chapters, applications communicate with each other using application protocols.

Some tools are created to interact with these protocols. For example, you can use Telnet to interact with the SMTP protocol to send emails or SCP to interact with the SSH protocol to transfer files.

In this chapter, you will learn how to program your own TCP clients and servers in Java.

This will allow you to create your own network applications, such as a chat server, a file server, a web server, etc.

TCP

TCP is a transport protocol. It is used to transfer data between two applications. TCP can only do UniCast: one application can only communicate with one other application.

TCP is a connection-oriented protocol: a connection must be established between the two applications before data can be exchanged in a bidirectional way.

TCP is a reliable protocol: data sent is guaranteed to be received by the other application.

A good analogy is to think of TCP as a phone call. You must first establish a connection with the other person before you can talk to them. Once the connection is established, you can talk to the other person and they will hear everything you say.

With the help of port numbers, TCP allows multiple applications to communicate with each other on the same machine.

TCP is a stream-oriented protocol: data is sent as a stream of bytes. The application must split the data into segments. Each segment is identified by a sequence number.

TCP segments are encapsulated in IP packets, called payloads.

Thanks to the sequence numbers, TCP is able to reassemble the segments in the correct order. If a segment is lost, TCP will retransmit it.

The Socket API

The Socket API is a Java API that allows you to create TCP/UDP clients and servers. It is described in the java.net package in the java.base module.

It has originally been developed in C in the context of the Unix operating system by Berkeley University. It has been ported to Java and is now available on many platform and languages.

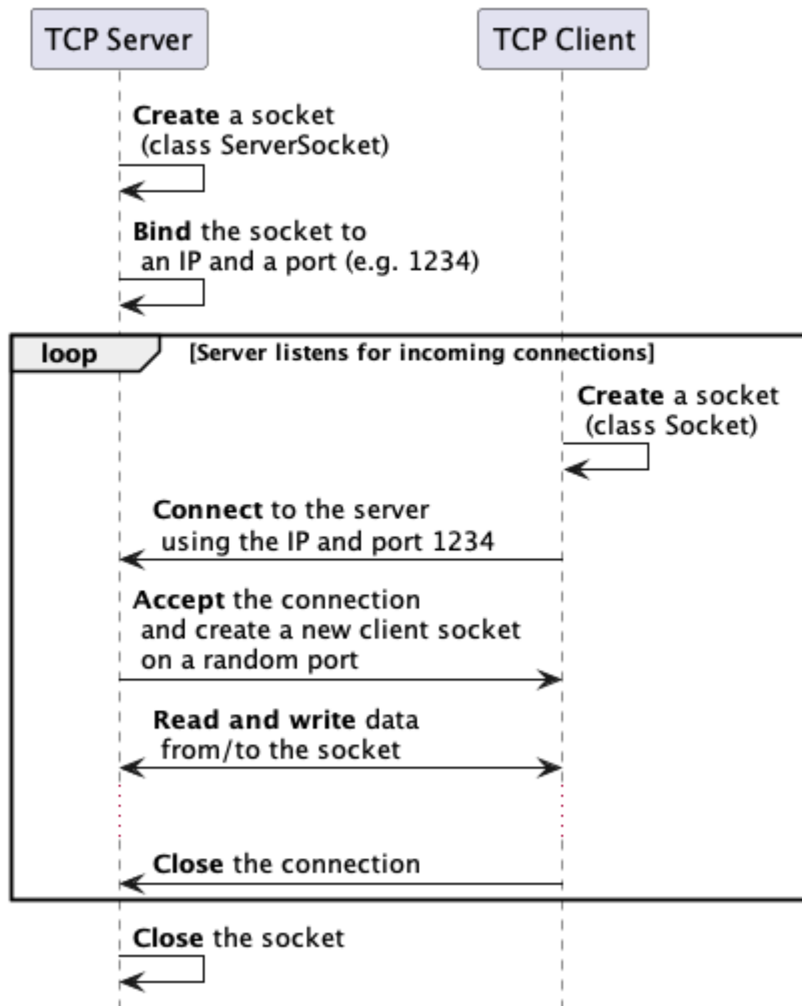
To make it simple, a socket is just like a file that you can open, read from, write to and close. To exchange data, sockets on both sides must be connected.

A socket is identified by an IP address and a port number.

A socket can act as a client or as a server:

- A socket accepting connections is called a server socket (class ServerSocket).
- A socket initiating a connection is called a client socket (class Socket).

The following schema shows the workflow of a client/server application:



Client/server common methods

Operation	Description
socket()	Creates a new socket
getInputStream()	Gets the input stream of a socket
getOutputStream()	Gets the output stream of a socket
close()	Closes a socket

Client workflow and methods

In order to create a client, the following workflow is followed:

> 1. Create a socket (class `Socket`)
> 2. Connect the socket to an IP address and a port number
> 3. Read and write data from/to the socket
> 4. Flush and close the socket

The available methods are the following:

Operation	Description
connect()	Connects a socket to an IP address and a port number

Server structure and methods

In order to create a server, the following workflow is followed:

> 1. Create a socket (class ServerSocket)
> 2. Bind the socket to an IP address and a port number
> 3. Listen for incoming connections
> 4. Loop
>   1. Accept an incoming connection - creates a new socket (class Socket) on a random port number
>   2. Read and write data from/to the socket
>   3. Flush and close the socket
> 5. Close the socket (ServerSocket)

The available methods are the following:

Operation	Description
bind()	Binds a socket to an IP address and a port number
listen()	Listens for incoming connections
accept()	Accepts an incoming connection

Processing data from streams

Sockets use data streams to send and receive data, just like files.

You get an input stream to read data from a socket and an output stream to write data to a socket.

```
// Get input stream
input = socket.getInputStream();

// Get output stream
output = socket.getOutputStream();
```

You can then decorate the input and output streams with other streams to process the data.

```
// Get input stream as text
input = new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8);

// Get output stream as text
output = new OutputStreamWriter(socket.getOutputStream(), StandardCharsets.UTF_8);
```

Use buffered streams to improve performance.

```
// Get input stream as binary with buffer
input = new BufferedReader(new InputStreamReader(socket.getInputStream());

// Get output stream as binary with buffer
output = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream());
```

Warning

Do not forget to flush the output stream after writing data to it. Otherwise, the remaining data in the buffer will not be sent to the other application!

```
out.flush();
```

Also, do not forget all the good practices seen in the Java IOs chapter (encoding, buffering, etc.). They must be applied here too!

Variable length data

Depending on the application protocol, the data sent can have a variable length.

There are two ways to handle variable length data:

- Use a delimiter
- Use a fixed length

If the data has a delimiter, you can use a buffered reader to read the data until the delimiter is found.

```
// End of transmission character
String EOT = "\u0004";

// Read data until the delimiter is found
String line;
while ((line = in.readLine()) != null && !line.equals(EOT)) {
  System.out.println(
    "[Server " + SERVER_ID + "] received data from client: " + line
  );
}
```

If the data has a fixed length, you must send the length of the data before sending the data itself.

```
// Send the length of the data
out.write("DATA_LENGTH " + data.length() + "\n");

// Send the data
out.write(data);

// Read the length of the data
String[] parts = in.readLine().split(" ");
int dataLength = Integer.parseInt(parts[1]);

// Read the data
for (int i = 0; i < dataLength; i++) {
  System.out.print((char) in.read());
}
```

Handling one client at a time

A server that handles one client at a time is called single-threaded, or single-threaded server.

A single-threaded server is quite simple to implement:

1. It creates a socket to listen for incoming connections.
2. When a connection is accepted, it creates a socket to communicate with the client.
3. It then reads the data sent by the client and sends a response.

The main drawback of a single-threaded server is that it can only handle one client at a time. If another client tries to connect, it will have to wait until the first client is disconnected.

An analogy is to think of a single-threaded server as a restaurant with only one table. If a customer is already sitting at the table, another customer will have to wait until the first customer leaves.

A single-threaded server is therefore not suitable for production. It is suitable for testing and learning purposes. In order to manage multiple clients, a server must handle multiple sockets.

Multiple ways exist to handle multiple sockets at the same time and is called concurrency.

Handling multiple clients with concurrency

Concurrency is the ability of an application to handle multiple clients at the same time.

There are multiple ways to handle multiple clients with concurrency (among others):

- Multi-processing
- Multi-threading
- Asynchronous programming

Java has a package for concurrency called java.util.concurrent.

In this course, we will focus on multi-threading but the other methods are equally valid and interesting to learn.

Multi-processing

Multi-processing is the ability of an application to handle multiple processes at the same time.

A process is a program in execution. It is identified by a process ID.

A process has its own memory space. It cannot access the memory space of another process.

The main process is the process that is created when the application starts.

It creates other processes to handle multiple clients.

A process is a heavy-weight object. It is quite expensive to create and destroy as it is a copy of the main process.

Processes can communicate with each other using inter-process communication (IPC) but it is quite complex to implement.

An analogy is to think of a process as restaurant chain with multiple restaurant. Each restaurant has only one table and can handle one customer. If a customer is already sitting at a table of a given restaurant, another customer can sit at a table at another restaurant.

Multi-threading

Multi-threading is the ability of an application to handle multiple threads at the same time.

A thread is a sequence of instructions that can be executed independently of the main thread.

The main thread is the thread that is created when the application starts.

It creates other threads to handle multiple clients.

Each thread has its own stack and its own program counter.

A thread is therefore quite similar to a process, except that it shares the same memory space as the other threads. It is therefore much cheaper to create and destroy than a process (but still more expensive than a simple object).

Threads can communicate with each other using shared memory.

Threads are more lightweight than processes but their number is limited by the operating system.

There are two ways to manage threads:

- Unlimited threads
- Thread pool that limits the number of threads

When discussing the unlimited threads approach, an analogy is to think of a restaurant with no tables at all. When a new customer arrives, the restaurant manager adds a new table for the customer. Each table can handle one customer.

Using this approach, the more customers arrive, the more tables are added. This approach is not suitable for production as space and resources are limited.

When discussing the thread pool approach, an analogy is to think of a restaurant with a limited number of tables. When a new customer arrives, the restaurant manager checks if a table is available. If a table is available, the customer can sit at the table. If no table is available, the customer will have to wait until a table is available.

Using this approach, the number of tables is limited. This approach is suitable for production as space and resources are managed and limited.

Asynchronous programming

Asynchronous programming is the ability of an application to handle multiple tasks at the same time, without blocking the main thread.

Using asynchronous programming, the main thread can perform other tasks while waiting for a task to complete.

Asynchronous programming is based on the concept of callbacks. A callback is a function that is called when a task is completed.

An analogy is to think of asynchronous programming as a food truck without any tables. Once a customer wants something to eat, the person managing the food truck gives the customer a ticket. The customer then waits until the food is ready but can do other things in the meantime.

Once the food is ready, the person managing the food truck calls the customer. The customer then comes to the food truck to get the food.

Asynchronous programming is quite complex to implement. It is therefore not covered in this course.

Node.js is a good example of asynchronous programming.

Practical content

Get the required files

In this section, you will retrieve the latest changes from the heig-vd-dai-course/heig-vd-dai-course-code-examples repository.

Get the latest changes from the code examples

Pull the latest changes from the previously cloned heig-vd-dai-course/heig-vd-dai-course-code-examples repository or clone it if you have not done it yet.

Explore the code examples

In the `12-java-tcp-programming` directory, checkout the `README.md` file to learn how to run the code examples.

Take some time to explore the code examples.

Send an email using a SMTP client written in Java with the Socket API

In this section, you will learn how to send an email using the SMTP protocol using the Java Socket API.

Start MailHog

Just as in the SMTP and Telnet chapter, start MailHog in order to receive the emails sent by the Java code examples.

Compile and run the SMTP client

In the `12-java-tcp-programming` directory, compile and run the `SmtpClientExample` code example.

```
# Compile the example
javac SmtpClientExample.java

# Run the example
java SmtpClientExample
```

The mail has been sent to the MailHog SMTP server. You can check it in the MailHog Web interface at http://localhost:8025.

Take some time to explore the code example. You should notice the commands are the same as the ones used with Telnet but in an automated way!

Run full client/server examples

Explore and run the code examples

In the `12-java-tcp-programming` directory, checkout the `README.md` file to learn how to run the code examples.

Take some time to explore the code examples. Run them and see what they do.

Note

Please be aware that the `TcpServerVirtualThreadTextualExample` example must be run with Java 21 or later. It is not mandatory to run this example but you must understand how it works.

This example is not compatible with Java 17 but is already available in the code examples repository for future use.

Answer the following questions

Using the official Java documentation, can you explain the differences between the following different implementations? When should you use one or the other and why?

- TcpServerSimpleTextualExample
- TcpServerSingleThreadTextualExample
- TcpServerMultiThreadTextualExample
- TcpServerCachedThreadPoolTextualExample

- `TcpServerFixedThreadPoolTextualExample`
- `TcpServerVirtualThreadTextualExample`

Are you able to explain why the `TcpServerSingleThreadTextualExample` does not work as expected?

Share your findings

Share your results in the GitHub Discussions of this organization: https://github.com/orgs/heig-vd-dai-course/discussions.

Create a new discussion with the following information:

- **Title**: DAI 2024-2025 - Concurrent Java TCP servers - First name Last Name
- **Category**: Show and tell
- **Description**: Answer the questions for this section. Add links to the official Java documentation to support your answers.

This will notify us that you have completed the exercise and we can check your work.

You can compare your solution with the official one stated in the Solution section, however, **we highly recommend you to try to complete the practical content by yourself first to learn the most**.

Go further

This is an optional section. Feel free to skip it if you do not have time.

- Based from the code examples, are you able to create a complete TCP client/server application in Java that implement the DAI protocol presented in chapter Define an application protocol? Feel free to create a new repository for this and share it with us in a new discussion on GitHub Discussions!

Conclusion

What did you do and learn?

In this chapter, you have learned how to use the Socket API to create your own TCP clients and servers in Java.

You have also learned how to handle multiple clients at the same time using concurrency.

You now have all the knowledge to create your TCP network applications. This is a big step forward!

You are now able to create your own network applications, such as a chat server, a file server, a web server, etc. Congratulations!

Test your knowledge

At this point, you should be able to answer the following questions:

- What is a socket?
- What is the difference between a server socket and a client socket?
- What is the purpose of concurrency?
- Cite three ways to handle multiple clients with concurrency.

Finished? Was it easy? Was it hard?

Can you let us know what was easy and what was difficult for you during this chapter?

This will help us to improve the course and adapt the content to your needs. If we notice some difficulties, we will come back to you to help you.

Note

Vous pouvez évidemment poser toutes vos questions et/ou vos propositions d'améliorations en français ou en anglais.

N'hésitez pas à nous dire si vous avez des difficultés à comprendre un concept ou si vous avez des difficultés à réaliser les éléments demandés dans le cours. Nous sommes là pour vous aider !

➡ GitHub Discussions

You can use reactions to express your opinion on a comment!

What will you do next?

You will start the practical work!

Additional resources

Resources are here to help you. They are not mandatory to read.

- *None yet*

Missing item in the list? Feel free to open a pull request to add it!

Solution

You can find the solution to the practical content in the heig-vd-dai-course/heig-vd-dai-course-solutions repository.

If you have any questions about the solution, feel free to open an issue to discuss it!

Sources

- Main illustration by Carl Nenzen Loven on Unsplash