Java TCP programming - 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.

Table of contents

Objectives

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

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.

Explore the code examples

Individually, or in pair/group, **take 10 minutes to explore and discuss the code examples** provided in the heig-vd-dai-course/heig-vd-dai-course-code-examples repository. Clone it or pull the latest changes to get the code examples.

The code examples are located in the `12-java-tcp-programming` directory.

Try to answer the following questions:

- How do the code examples work?
- What are the main takeaways of the code examples?
- What are the main differences between the code examples?

You can use the following theoretical content to help you.

TCP

TCP is a transport protocol. It is used to transfer data between two applications.

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 can only do unicast: one application can only communicate with one other application at the same time.

It is considered as a reliable protocol as 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. If they did not hear you well, you can repeat what you said until they hear you. They can, of course, also talk to you.

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 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.

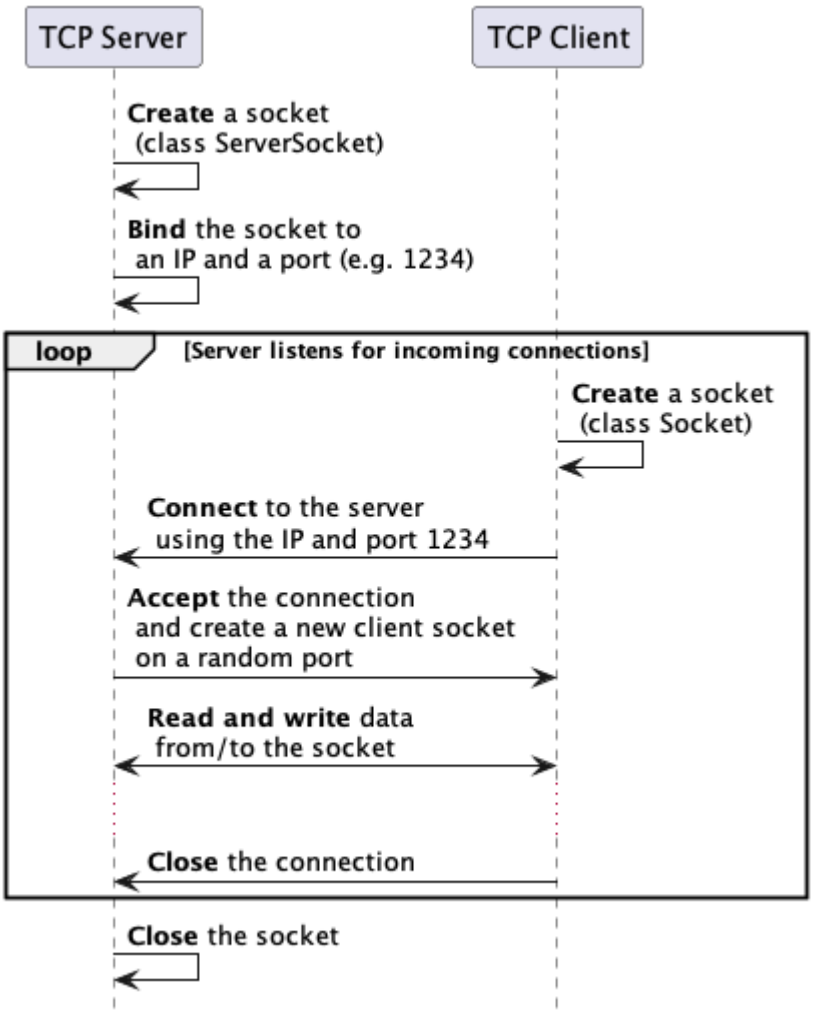
To make it simple, a socket is just like a file that you can open, read from, write to and close. To exchange data, sockets on both sides must be connected. The processing is the same as with files, seen in the Java IOs chapter.

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, just as with IOs.

```
// 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()));
```

Important

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] 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());
}
```

Read-eval-print loop (REPL)

In order to run multiple commands/actions on the server without closing the connection, you can use what is called a read-eval-print loop (REPL).

To make it simple, a REPL is simply a loop that will ask the user to input commands. The loop will then execute the command and display the result. The loop will continue until the user decides to exit the loop.

In the context of a server, the server will wait for the client to send a command. The server will then execute the command and send the result back to the client. The server will continue to wait for the client to send a new command without closing the connection.

On the client side, the client can interact with the server by sending commands to the server until they decide to close the connection.

Both the client and the server can close the connection at any time. It is up to the developer to decide when and who manage to close the connection.

Practical content

Execute the code examples

Return to the code examples and take some time to execute them, understand them and see the results.

Update your application protocol

Now that you have gained new knowledge regarding TCP, update the application protocol you have created for the *"Guess the number"* game in the Define an application protocol chapter chapter to reflect the usage of the TCP protocol.

You can check the official solution in the Define an application protocol chapter.

Try to access the server from multiple clients at the same time

Try to access the server from multiple clients at the same time (start the client multiple times). You will see that the server can only handle one client at a time.

Do you have any idea why? You will find the answer in a future chapter but you can try to find it by yourself now. Discuss with your peers if needed to share your findings.

Explore the Java TCP programming template

In this section, you will explore the Java TCP programming template.

This is a simple template that you can use to create your own TCP clients and servers in Java.

The template is located in the heig-vd-dai-course/heig-vd-dai-course-java-tcp-programming-template.

Take some time to explore the template. Then, try to answer the following questions:

- How would you use it to create your own TCP clients and servers?
- What are the main takeaways of the template?
- How you would you implement a TCP network application using the template and the provided code examples?

You can use the template to create your own TCP network applications.

Go further

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

Implement the *"Guess the number"* game

Implement the *"Guess the number"* game using the application protocol you have made from the Define an application protocol chapter.

You can use the application protocol you have made or the one provided in the solution if you have not done it.

Use the template and the code examples you just explored to help you implement the game.

When you create a new repository, you can choose to use a template. Select the heig-vd-dai-course/heig-vd-dai-course-java-tcp-programming-practical-content template.

Warning

Please make sure that the repository owner is your personal GitHub account and not the heig-vd-dai-course organization.

Dockerize the application

Using the Docker knowledge you have acquired in the Docker and Docker Compose chapter, dockerize the application.

The steps to dockerize the application are the following:

- Create a Dockerfile for the application
- Publish the application to GitHub Container Registry

You should then be able to run the server and the client in Docker containers and access the server from the client using the following commands:

```
# Start the server
docker run --rm -it --name the-server <docker-image-tag> server

# Start the client and access the server container
docker run --rm -it <docker-image-tag> client --host the-server
```

Note

I (Ludovic) was not able to test these commands thoroughly. You might need to adapt them to make them work. If something does not work, feel free to tell me so I can update the commands.

The --name sets the name of the container as well as the hostname of the container. This allows to access the server container using its hostname from the client.

You might notice that no ports are published to the host. As both container run on Docker, they share the same network bridge. They can thus communicate together without passing by the host.

Compare your solution with the official one

Compare your solution with the official one stated in the Solution section.

If you have any questions about the solution, feel free to ask as described in the Finished? Was it easy? Was it hard? section.

Go one step further

- Can you update the network application to allow the client to specify the range of the number to guess before starting the game?
- Can you implement the "*Temperature monitoring*" application with TCP?

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.

Congratulations! It 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.

As for now, only one client can access the server at the same time. You will see in a future chapter how to manage multiple clients at the same time!

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?
- How do sockets compare to files?
- Why is TCP considered as a reliable protocol?

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?

In the next chapter, you will learn the following topics:

- Java UDP programming
    - How does it compare to TCP?
    - How to create efficient UDP network applications
    - Implement the *"Temperature monitoring"* application using UDP (optional)

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