Java UDP 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

- Table of contents
- Objectives
- Explore the code examples
- UDP
- Differences between TCP and UDP
- UDP datagrams
- Reliability
- UDP in the Socket API
- Unicast, broadcast and multicast
  - Unicast
  - Broadcast
  - Multicast
- Messaging patterns
- Service discovery protocols
- Practical content
  - Update your application protocol
  - Learn to use the debugger
  - Try to emit from multiple emitters at the same time
  - Explore the Java UDP programming template
  - Go further
  - Compare your solution with the official one
- 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

You have seen and experimented with TCP in the previous chapter. You have seen that TCP is a connection-oriented protocol. It means that a connection must be established before sending data.

In this chapter, you will see and experiment with UDP. UDP is mainly used when reliability is not required. It is used for streaming, gaming, etc.

UDP is sensibly different from TCP and it is important to understand the differences between the two protocols.

Explore the code examples

Individually, or in pair/group, **take 15 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 `13-java-udp-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.

UDP

UDP is a transport layer protocol, like TCP. It is used to send data over the network. However, there are numerous differences between TCP and UDP.

UDP is a connectionless protocol, which means that it does not require to establish a connection before sending data.

UDP does not provide any reliability mechanism. It does not guarantee that the data will be received by the receiver at all, nor that the data will be received in the same order as it was sent.

A good analogy is to think of UDP as the postal service with postcards: you write multiple postcards and send them to someone. You do not know if the postcards will be received nor if they arrive in the same order as they were sent. You do not know if the postcards will be received at all if the postal service loses them.

Just as with postcards, UDP is used when reliability is not required.

Differences between TCP and UDP

The following table summarizes the differences between TCP and UDP.

TCP	**UDP**
Connection-oriented	Connectionless
Reliable	Unreliable
Stream protocol	Datagram protocol
Unicast	Unicast, broadcast and multicast
Request-response	Fire-and-forget, request-response (manual)
-	Service discovery protocols
Used for FTP, HTTP, SMTP, SSH, etc.	Used for DNS, streaming, gaming, etc.

UDP datagrams

Unlike TCP, UDP is not a stream protocol. It is a datagram protocol. It means that UDP sends data in discrete chunks called datagrams.

Datagrams are like the postcards in the previous analogy. They are sent independently from each other. They are not related to each other. They contain a destination address, a payload and the sender address. If you need to, you can use the sender address to reply to the sender.

UDP datagrams are composed of a header and a payload. The header contains information about the datagram, such as the source and destination port. The payload contains the data to send.

The size of the payload is limited to 65,507 bytes. It is because the payload length is encoded on 16 bits in the header.

The payload of a UDP datagram can be a notification, a request, a query, a response, etc. It is up to the application to define the payload format.

If the payload is too large, the datagram will be fragmented. It means that the payload will be split into multiple datagrams. The receiver will have to reassemble the datagrams to get the original payload.

Reliability

As UDP does not provide any reliability mechanism, it is up to the application to implement it. For example, the application can implement a mechanism to acknowledge the reception of a datagram and retransmit it if it was not received.

What is offered by TCP has to be implemented by the application with UDP.

In certain cases, reliability is not required. Some applications are tolerant to data loss.

For example, streaming can be a perfect use case for UDP. If a datagram is lost, it does not matter much: the receiver will receive the next datagram and the stream will continue. A good example would be the streaming of a live event on your television:

- If a few datagrams are lost, the receiver might notice it with a few glitches (video artifacts) but it will not affect the entire stream.
- If too many datagrams are lost, the receiver will not be able to reassemble the payload and the stream will stop.

Some video services such as Jitsi (an open source Zoom/Google Meet/Teams alternative) can make usage of the UDP protocol with the help of WebRTC. However, even these applications might prefer to make usage of the TCP protocol to guarantee the reliability of their services.

The game Factorio (a game where you build and manage factories) is another good example of an application that make usage of the UDP protocol when playing on a multiplayer server:

> Factorio uses UDP only. The game builds its own "reliable delivery" layer built on UDP to deal with packet loss and reordering issues.
>
> https://wiki.factorio.com/Multiplayer

As mentioned before, it is up to the application to implement a reliability mechanism if required (with a message ID and an acknowledgement for example, just as TCP).

We can illustrate this with the following example:

You have developed a very simple application protocol where clients can send INCREMENT and DECREMENT commands to increment/decrement a counter on the server. The counter is shared between all clients.

If the clients send 10 INCREMENT commands, the counter should be incremented by 10.

In a perfect world, the server would receive 10 INCREMENT commands and the counter would be incremented by 10.

However, we know one of the datagrams could be lost. If the server receives 9 INCREMENT commands, the counter will be incremented by 9 instead of 10.

Both parties (the client and the server) could implement a reliability mechanism to solve this issue.

The server could implement a reliability mechanism to acknowledge the reception of a datagram. If a client does not receive an acknowledgement *within a specific period*, it should retransmit the datagram.

However, even the acknowledgement could be lost. The client could retransmit the datagram multiple times and the server could receive it multiple times.

The server could implement a mechanism to detect duplicate datagrams and ignore them. It could also implement a mechanism to detect out-of-order datagrams and reorder them.

Handling reliability is quite challenging. In the context of this course, reliability is not required. We will focus on the UDP protocol itself and not on the reliability mechanism(s).

If you are interested, you can have a look at the Automatic Repeat reQuest (ARQ) protocol. It is a mechanism used to detect and retransmit lost datagrams.

UDP in the Socket API

As seen in the Java TCP programming chapter, 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.

In the UDP world, the `Socket` class is replaced by the `DatagramSocket` class.

The `DatagramSocket` class is used to create UDP clients and servers. It is used to send and receive UDP datagrams.

A datagram is created with the `DatagramPacket` class. It is used to create a datagram with a payload and a destination address.

A multicast socket is created with the `MulticastSocket` class. It is used to create a multicast datagram with a payload and a multicast address, allowing multiple hosts to receive the datagram.

UDP can be used to create a client-server architecture. However, it is not required. It is possible to create a peer-to-peer architecture with UDP.

Unicast, broadcast and multicast

Unlike TCP, UDP supports three types of communication: unicast, broadcast and multicast (TCP only supports unicast).

Unicast

Unicast is the most common type of communication. It is a one-to-one communication. It means that a datagram is sent from one host to another host, just like TCP.

Think of it as a private conversation between two people.

To send a unicast datagram, the sender must know the IP address and port of the receiver. It is mostly the same as TCP, without all the features provided by TCP but all the performance of UDP.

Broadcast

Broadcast is a one-to-all communication. It means that a datagram is sent from one host to all hosts on the network.

Think of it as a public announcement.

To send a broadcast datagram, the sender must know the broadcast address. The broadcast address is a special address that represents all hosts on the network and/or all hosts of a specific subnet.

The broadcast address is defined by the subnet mask. The subnet mask is a 32-bit number. It is represented as four numbers separated by a dot (e.g. `255.255.255.0`). Sometimes, the subnet mask is represented as a single number (e.g. `/24` for `255.255.255.0` as 24 bits are set to `1`).

A good example is stated in the following table (source: https://en.wikipedia.org/wiki/Broadcast_address):

Network IP address breakdown for 172.16.0.0/12	**Binary form**	**Dot-decimal notation**
1. Network IP Address	10101100.0001**0000.00000000.00000000**	172.16.0.0
2. Subnet Mask, or just "Netmask" for short (The /12 in the IP address in this case means only the left-most 12 bits are 1s, as shown here. This reserves the left 12 bits for the network address (prefix) and the right 32 - 12 = 20 bits for the host address (suffix).)	11111111.1111**0000.00000000.00000000**	255.240.0.0
3. Bit Complement (Bitwise NOT) of the Subnet Mask	00000000.0000**1111.11111111.11111111**	0.15.255.255
4. Broadcast address (Bitwise OR of *Network IP Address* and	10101100.0001**1111.11111111.11111111**	172.31.255.255

Network IP address breakdown for 172.16.0.0/12	**Binary form**	**Dot-decimal notation**
Bit Complement of the Subnet Mask. This makes the broadcast address the *largest possible IP address (and host address, since the host address portion is all 1s) for any given network address.*)		

If you want to send a broadcast to all devices on all network subnets, you can use the 255.255.255.255 broadcast address.

Important

You must be aware that there can be restrictions on the use of broadcast. For example, broadcast is limited to the local network but can still be blocked by a firewall and/or a router.

Multicast

Multicast is a one-to-many communication. It means that a datagram is sent from one host to multiple hosts.

Think of it as a group conversation.

To send a multicast datagram, the sender uses a multicast address. The multicast address is a special address that represents a group of hosts on the network. Think of it as a radio channel or a Discord channel: everyone on the channel will receive the messages sent in a given channel.

Multicast addresses are specific IP addresses in the range from `224.0.0.0` to `239.255.255.255` for IPv4 and `f00::/8` for IPv6.

Just as for ports, some multicast addresses are reserved for specific purposes. A complete list is available on the IANA website and further described in the RFC 5771.

For local networks, the multicast range is from the **Administratively Scoped Bloc** of the RFC. More details are available in the RFC 2365.

Any multicast addresses in the range `239.0.0.0` to `239.255.255.255` can be used for your own applications.

Just as for broadcast, the sender must know the multicast address to send a datagram to a multicast group. Just as for broadcast as well, there can be restrictions on the use of multicast.

Multicast is quite guaranteed **not** to work on the public Internet. It is only guaranteed to work on a local network. If you need to use multicast between multiple networks, you must use a tunnel such as a virtual private network (VPN) to bypass this restriction.

Multicast is presented in this course because it is an important concept in service discovery protocols. However, you must be aware that it is quite not possible to use multicast on the public Internet, thus it greatly limits its usage.

Also, Multicast is a complex topic. It is not covered in depth in this course. For a deeper understanding of possible usages of multicast on the Internet, you can read the following resources:

- IP multicast
- Internet Group Management Protocol
- Internet Protocol television

Messaging patterns

As UDP does not provide a connection mechanism, it is up to the application to define the messaging pattern (how to send and receive data).

There are two common messaging patterns: fire-and-forget and request-response.

The fire-and-forget pattern is the simplest messaging pattern. It is a one-way communication. It means that a datagram is sent from one host to another host without expecting a response.

The fire-and-forget pattern is used when the sender does not need to know if the datagram was received or not.

The request-response (sometimes called request-reply) pattern is a two-way communication. It means that a datagram is sent from one host to another host and a response is expected.

When creating a datagram, it is possible to specify a port. While not mandatory, this port can be used by the receiver to know whom to reply to.

If no port is specified, the operating system will simply assign a random port for the out going datagram.

The receiver of the datagram can then extract the sender's IP address and port and use them to reply to the sender using unicast.

The request-response pattern can be used when the sender needs to know if the datagram was received or not.

Both sides of the communication can send a request and receive a response.

Service discovery protocols

With unicast, the sender must know who the receiver is; the sender must know the IP address of the receiver.

With broadcast and multicast, the sender does not need to know who the receivers are; the sender does not need to know the IP address of the receivers. The sender knows that nodes nearby (or those who expressed interest in the broadcast) will receive the datagram.

Using this property, it is possible to create service discovery protocols.

Service discovery protocols are used to discover services on the network. They are used to find services without knowing their IP address.

There are two types of service discovery protocols: passive and active.

Passive service discovery protocols are based on broadcast or multicast. They are used to announce the presence of a service on the network.

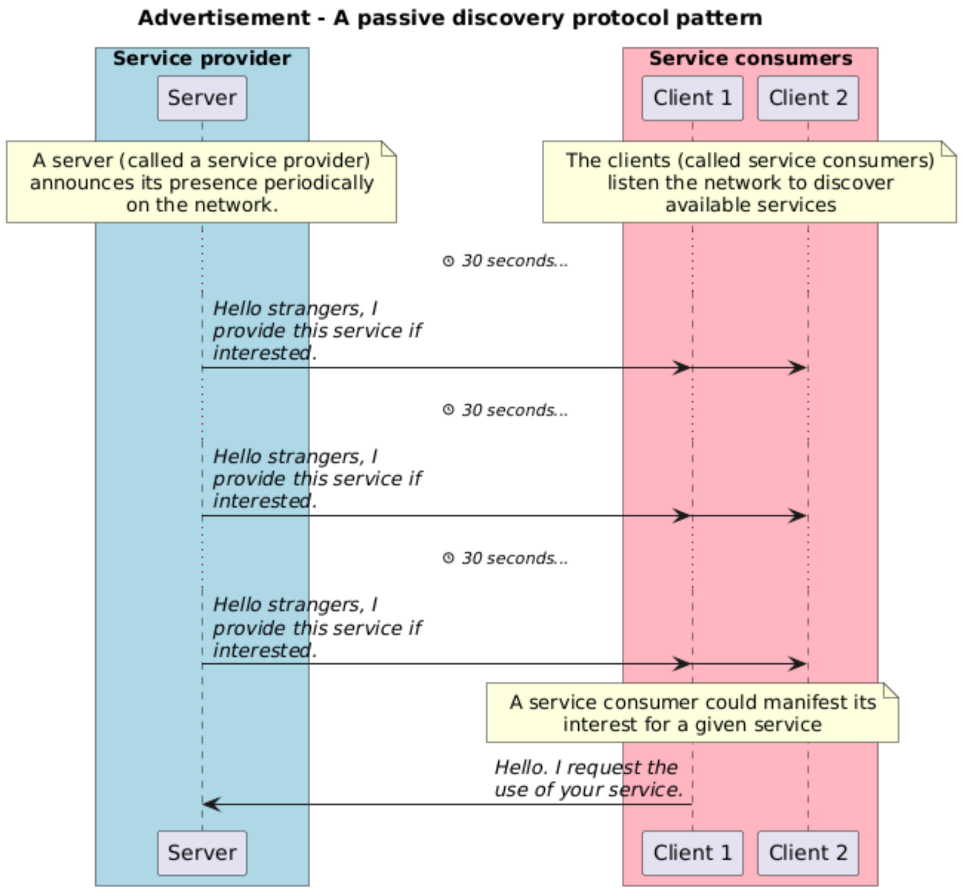
Active service discovery protocols are also based on broadcast or multicast but then switch to unicast. They are used to query the network to find a service.

There are many service discovery protocol patterns. The most common are the following:

- Advertisement - A passive discovery protocol pattern: a server (called a service provider) announces its presence on the network. The service provider sends a broadcast or multicast datagram to announce its presence. The datagram contains information about the service (name, IP address, port, etc.). The datagram is sent periodically to announce that the service is still available.

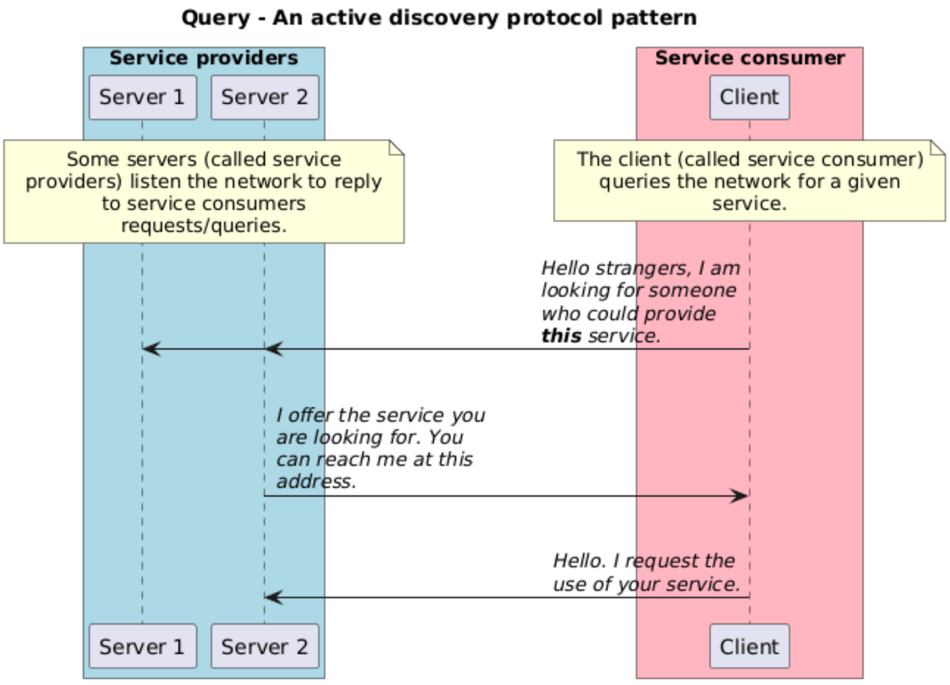
  The clients (called service consumers) listen to the broadcast or multicast datagrams to discover the services on the network.

  If a service consumer is interested by the service provider announcement, it can manifest its interest.


Advertisement - A passive discovery protocol pattern
Service provider
Server
Service consumers
Client 1
Client 2
A server (called a service provider) announces its presence periodically on the network.
The clients (called service consumers) listen the network to discover available services
30 seconds...
Hello strangers, I provide this service if interested.
30 seconds...
Hello strangers, I provide this service if interested.
30 seconds...
Hello strangers, I provide this service if interested.
A service consumer could manifest its interest for a given service
Hello. I request the use of your service.
Server
Client 1
Client 2

- Query - An active discovery protocol pattern: a client (called a service consumer) queries the network to find a service. The client sends a unicast datagram on the network to request information about a service.

  If a service that provides the requested service (called a service provider) is available, it replies with a unicast datagram containing the requested information to connect to the service, just as seen with the request-response messaging pattern.


Query - An active discovery protocol pattern
Service providers
Server 1
Server 2
Service consumer
Client
Some servers (called service providers) listen the network to reply to service consumers requests/queries.
The client (called service consumer) queries the network for a given service.
Hello strangers, I am looking for someone who could provide **this** service.
I offer the service you are looking for. You can reach me at this address.
Hello. I request the use of your service.
Server 1
Server 2
Client

These patterns can still be used with other protocols such as TCP.

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 UDP, update the application protocol you have created for the *"Temperature monitoring"* application in the Define an application protocol chapter chapter to reflect the usage of the UDP protocol.

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

Learn to use the debugger

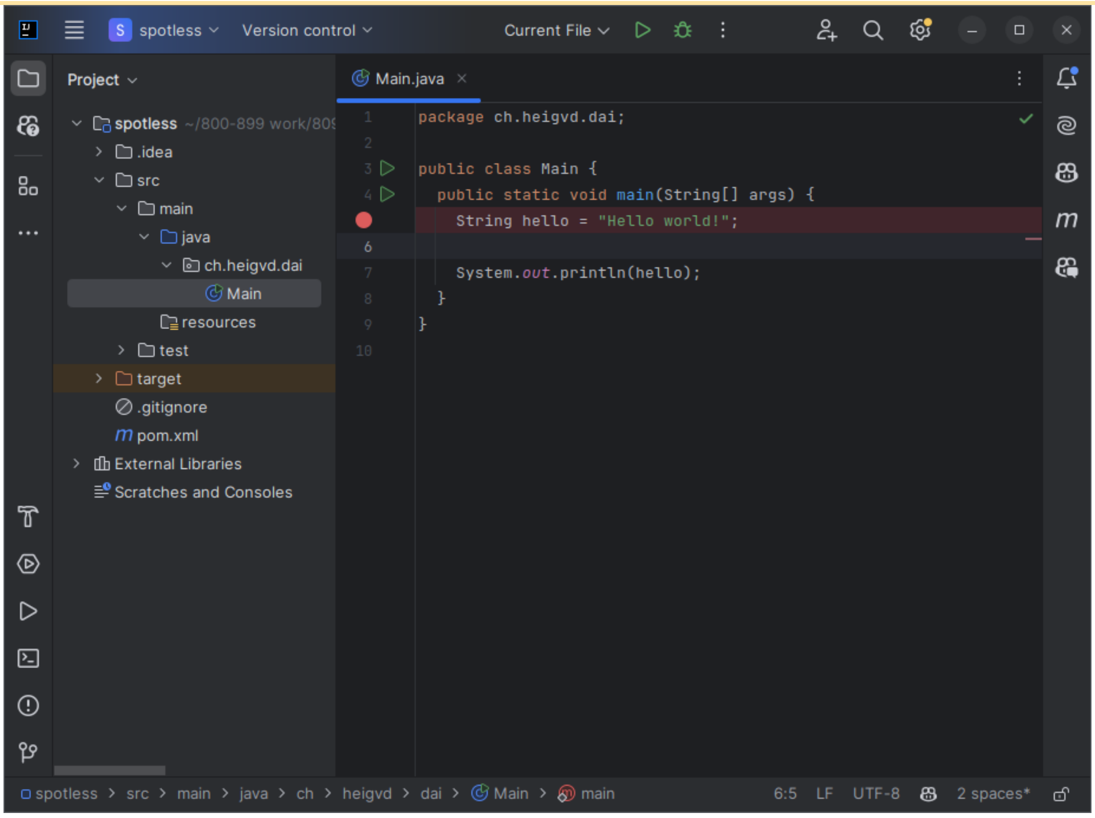
Every decent IDE has a debugger. The debugger is a tool that allows you to inspect the state of your program at runtime.

The debugger allows you to:

- Set breakpoints: a breakpoint is a point in your code where the program will stop when it is reached.
- Step through your code: you can step through your code line by line to see what is happening.
- Inspect variables and expressions: you can inspect the value of variables and expressions at runtime.

The debugger is a powerful tool to understand what is happening in your program.

In IntelliJ IDEA, set a breakpoint by clicking on the left side of the line number, as shown in the following screenshot:

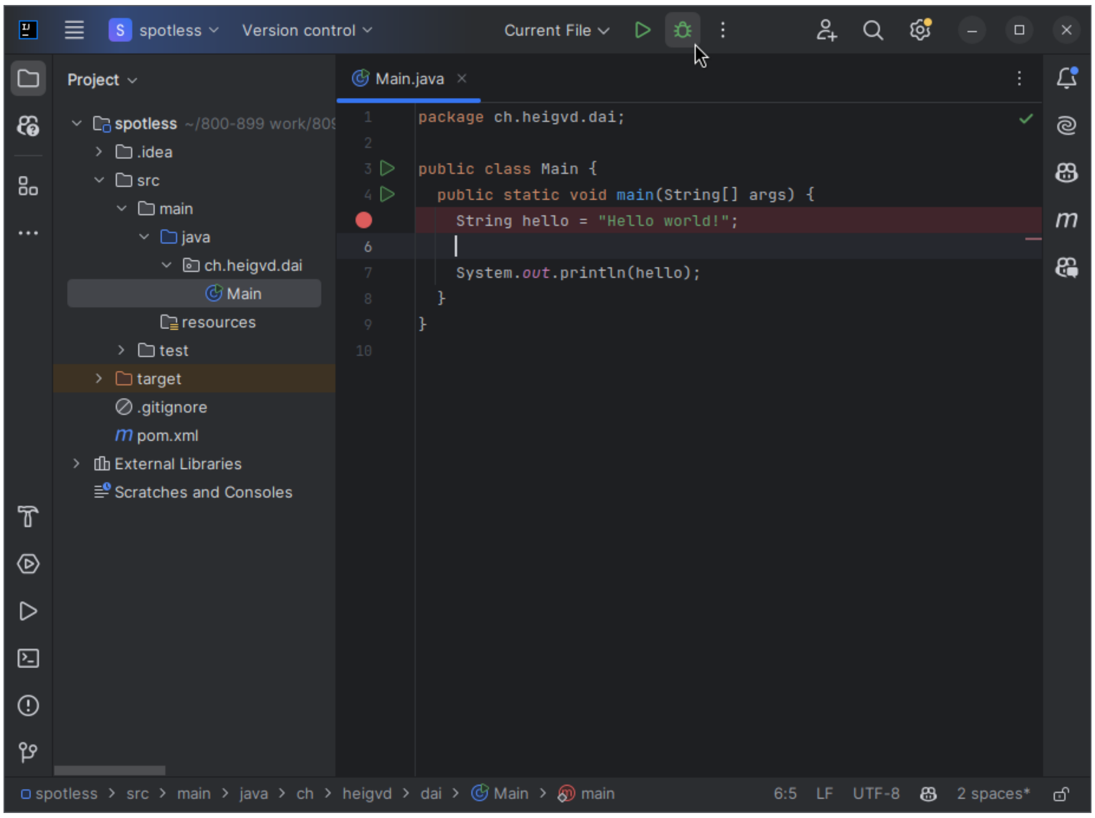


```java
package ch.heigvd.dai;

public class Main {
  public static void main(String[] args) {
    String hello = "Hello world!";

    System.out.println(hello);
  }
}
```

Then, run the program in debug mode by clicking on the bug icon, as shown in the following screenshot:

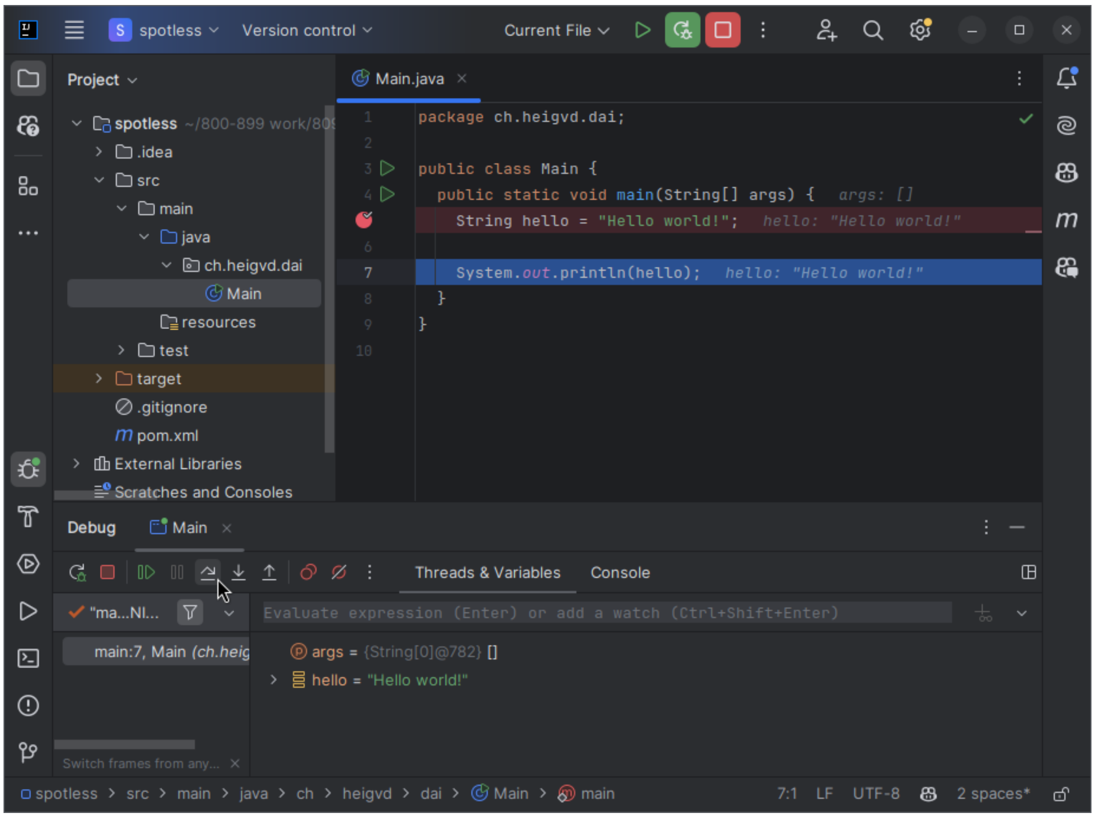


```java
package ch.heigvd.dai;

public class Main {
  public static void main(String[] args) {
    String hello = "Hello world!";

    System.out.println(hello);
  }
}
```

The program will stop at the breakpoint. You can then use the debugger to step through the code, inspect variables and expressions, etc., as shown in the following screenshot:



```
package ch.heigvd.dai;

public class Main {
  public static void main(String[] args) {   args: []
    String hello = "Hello world!";   hello: "Hello world!"

    System.out.println(hello);   hello: "Hello world!"
  }
}
```

```
args = {String[0]@782} []
hello = "Hello world!"
```

Take some time to learn how to use the debugger in your favorite IDE, this is a very useful tool.

Try to emit from multiple emitters at the same time

Try to emit from multiple emitters at the same time (start the emitter multiple times). You will see that the server will receive all messages from the emitters.

Do you have any idea why? How does it compare to the TCP examples you have seen in the Java TCP programming chapter?

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 UDP programming template

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

This is a simple template that you can use to create your own UDP emitters/clients and receivers/servers in Java.

The template is located in the heig-vd-dai-course/heig-vd-dai-course-java-udp-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 UDP emitters/clients and receivers/servers?
- What are the main takeaways of the template?
- How you would you implement a UDP network application using the template and the provided code examples?

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

Go further

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

Implement the *"Temperature monitoring"* application

Implement the *"Temperature monitoring"* 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-udp-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 emitter, the receiver and the operator in Docker containers and access the receiver from the operator using the following commands:

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

# Start the receiver
docker run --rm -it --name the-receiver <docker-image-tag> receiver --network-interface
               eth0

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

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 receiver container using its hostname from the operator.

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 application protocol to allow the operator to have the latest temperature for a given room or the average temperature of that room?

  Tip: this will require to store all the temperatures received for a given room and to calculate the average temperature instead of storing only the latest temperature.
- Are you able to Dockerize this application as well?

Conclusion

What did you do and learn?

In this chapter, you have learned how to use the UPD protocol to build different kind of network applications and the differences between TCP and UDP.

Using Java and Docker and Docker Compose, you were able to containerize your network application to use it anywhere.

Just as with TCP, you have now all the knowledge to build bigger and better network applications. We continue our journey toward network application programming.

Test your knowledge

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

- What are the differences between UDP and TCP?
- Why is UDP unreliable? How to mitigate this?
- What is a datagram? How can a datagram be sent without a server listening?
- What are the differences between unicast, broadcast and multicast?
- What are the messaging protocols and their differences?
- What are the service discovery protocols? How do they compare to each other?

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:

- Understand network concurrency.
- Manage multiple clients with concurrency.

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 Possessed Photography on Unsplash