Sockets

Sockets allow applications to communicate over a network, whether they are on the same machine or in different parts of the world.

This makes it possible to create distributed applications such as web servers, databases, chat rooms, and more.

What is a socket?

Each computer on the Internet has an address to uniquely identify it and allow other computers to interact with it.

Suppose you need to write a network application such as an instant messenger or an online game.

For this purpose, you need to organise interaction between multiple users of your application. You can implement this interaction using sockets.

A **socket** is an interface to send and receive data between different processes (running programs) in bidirectional order.

It is determined by the combination of the computer address in the network (e.g., 127.0.0.1, which is your own machine) and a port on this machine.
A port is an integer number from 0 to 65535, preferably greater than 1024 (e.g., 8080, 32254 and so on). So, a socket may have the following address: 127.0.0.1:32245.

To start the communication, one program called a **client** creates a socket to request a connection with another program called a **server**.

To this end, the client uses the server machine address and a specific port on which the server listens for incoming requests.

The server waits for new requests and accepts them or not.
If a request is accepted, the server creates a socket for interaction with the client. Both the client and server can send data to each other using their sockets.

As a rule, one server interacts with multiple clients. How long the interaction continues depends on the application.

Note: both the client and server programs can be on the same computer or on different machines connected through a network.

Sockets in Java

The Java Class Library provides two main classes for the interaction between programs using sockets:

- The Socket class represents one side of a two-way connection (used by clients and servers).
- The ServerSocket class represents a special type of socket that listens for and accepts connections to clients (only used by servers).

Both classes are located in the java.net.* package.

https://ucarecdn.com/ecdf0def-35b5-40f3-9279-0d13e84e0816/

So, a client program uses a Socket to send a connection request. The server has a ServerSocket to accept the request and then creates a new Socket for interaction with the client.

As an example, we will write a small **echo server**, which gets messages from clients and then sends them back. As a result, we will have two programs: a **client** and a **server**. Each program has its own main method.

The server-side code

First, let's consider the server-side code. To create a server socket, we use the following statement:

ServerSocket server = new ServerSocket(34522);

The server object listens on port 34522 for connection requests from clients.

The server can accept a new client and create a socket to interact with it:

Socket socket = server.accept(); // a socket to interact with a new client

The accept method forces the program to wait for a new client, i.e., it executes until a new client comes.

After this statement, we have a socket object that can be used to interact with the client.

To send and receive data, we need input and output streams:
DataInputStream input = new DataInputStream(socket.getInputStream());
DataOutputStream output = new DataOutputStream(socket.getOutputStream());

- The invocation input.readUTF() receives a string message from the client;
- The invocation output.writeUTF(message) sends a string message to the client.

If you need to send or receive something like movies or audio files, you may work directly with bytes rather than strings.

The client-side code

To start interacting with a server, we need at least one client.

First, we should create a socket, specifying the path and port of a server.

Socket socket = new Socket("127.0.0.1", 23456); // address and port of a server

Here, we use the **localhost** address ("127.0.0.1") as an example.

In a real case, your server will be hosted on another computer. Therefore, it is a good practice to take the address and port from an external configuration or command-line arguments.

To send and receive data to the server, we need input and output streams:
DataInputStream input = new DataInputStream(socket.getInputStream());
DataOutputStream output = new
DataOutputStream(socket.getOutputStream());

To demonstrate our client program, we first need to start the server and then run one or more client programs.
> Hello!
Received from the server: Hello!
Another example:
> What?
Received from the server: What?
The symbol > is not part of the input. It just marks the input line.

Serving multiple long-connected clients

If we want to develop a chat or a game server, our clients will not stop after sending a single message. They will periodically send and receive messages to/from the server.

Let's try to improve our client to send five messages to the **echo server**. Here is the modified client code that should be placed inside the try statement:
```
for (int i = 0; i < 5; i++) {
    Scanner scanner = new Scanner(System.in);
```

```
    String msg = scanner.nextLine();

    output.writeUTF(msg);
    String receivedMsg = input.readUTF();

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

The server was also modified to accept all five messages from a client.

```
for (int i = 0; i < 5; i++) {
    String msg = input.readUTF(); // read the next client message
    output.writeUTF(msg); // resend it to the client
}
```

> Hello1
Received from the server: Hello1
> Hello2
Received from the server: Hello2
> Hello3
Received from the server: Hello3
> Hello4
Received from the server: Hello4
> Hello5
Received from the server: Hello5

But if we start two or more clients, we will notice a strange effect. The server will not interact with the second client before responding to all messages from the first client. How come? Because we use only a single thread to process messages from all clients.

Multithreaded server

The simplest way to work with multiple clients simultaneously is to use multithreading!

Let one server thread (e.g., main) accept new clients while others interact with already accepted clients (one thread per client).

It is important to note that we do not use **try-with-resources** for server.accept() because we create a socket in one thread and close it in another. In this case, **try-with-resources** would be very error-prone since one thread could close the socket while another wants to use it.