Theory: Sockets

19 minutes 0 / 2 problems solved

§1. What is a socket?

As you may know, 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 organize interaction between multiple users of your application. You can make it 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` that is your own machine) and a port on this machine. A port is an integer number from `0` to `65535`, but 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 a communication, one program called **client** requests a connection with another program called **server** using machine address and a specific port on which the server is listening for incoming requests. The server just waits for a new request and accepts it or not. If the connection was 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 programs client and server can be on the same computer, or on different machines connected through a network.

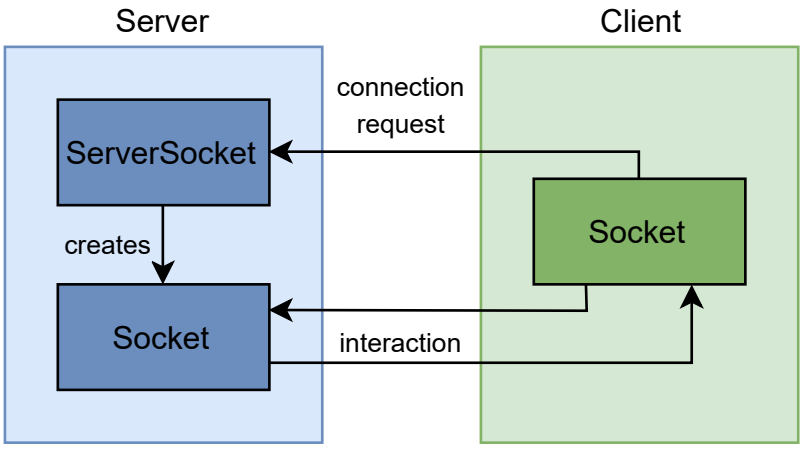
§2. Sockets in Java

Java Class Library provides two main classes to interact between programs using sockets:

- `Socket` represents one side of a two-way connection (used by clients and servers);
- `ServerSocket` represents a special type of sockets which listen for and accept connections to clients (only used by servers).

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

The following picture demonstrates when we should use both classes.



Socket and ServerSocket

So, a client program uses one `Socket` to send a connection request. The server has `ServerSocket` to accept the request and then it 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: **client** and **server**. Each program has its own `main` method to run.

Current topic:

Sockets

Topic depends on:

- ✕ Custom threads
- ✕ Exceptions in threads
- ✕ Output streams
- ✕ Input streams

Table of contents:

§3. Writing server-side code

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

```
ServerSocket server = new ServerSocket(34522);
```

The `server` object listens to the 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 to wait for a new client, i.e. it executes until a new client comes. As a result, we have a socket object that is 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 and receive something like movies or audio files, you may work with bytes directly rather than with strings.

Below we give you the full server program that accepts clients in a loop and just resends messages to them.

```
import java.io.*;
import java.net.*;

public class EchoServer {
    private static final int PORT = 34522;

    public static void main(String[] args) {
        try (ServerSocket server = new ServerSocket(PORT)) {
            while (true) {

                try (


                Socket socket = server.accept(); // accepting a new client


                DataInputStream input = new DataInputStream(socket.getInputStr
eam());


                DataOutputStream output = new DataOutputStream(socket.getOutpu
tStream())

                ) {

                    String msg = input.readUTF(); // reading a message

                    output.writeUTF(msg); // resend it to the client

                }

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

As you may see, this code is simple enough but it clearly demonstrates the basic ideas of socket communication. It creates only a single `ServerSocket` and then accepts connections from clients in an infinite loop. The program stops when a program shut down.

Pay attention, we also remember about exceptions and handle them in a simplified way. Here we also use the **try-with-resources** statement because the socket and streams can be closed automatically to avoid resource leaks.

§4. Writing client-side code

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

First, we should create a socket, specifying a path to the server.

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

We use the **localhost** address (`"127.0.0.1"`) just for an example. In a real case, your server is hosted on another computer. So, it is 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());
```

We have considered above how to use them.

Below we give you the full client program that connects to a server, sends one message and prints the received message from the server.

```
import java.io.*;
import java.net.Socket;
import java.util.Scanner;

public class EchoClient {
    private static final String SERVER_ADDRESS = "127.0.0.1";
    private static final int SERVER_PORT = 34522;

    public static void main(String[] args) {

        try (

            Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT);


    DataInputStream input = new DataInputStream(socket.getInputStream());


    DataOutputStream output  = new DataOutputStream(socket.getOutputStream
())

        ) {

            Scanner scanner = new Scanner(System.in);

            String msg = scanner.nextLine();



            output.writeUTF(msg); // sending message to the server

            String receivedMsg = input.readUTF(); // response message



            System.out.println("Received from server: " + receivedMsg);

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

Like before, we use **try-with-resources** to close the socket and streams.

To demonstrate our client program, first, we need to start the server and then run one or more client programs.

```
> Hello!
Received from server: Hello!
```

Another example:

```
> What?
Received from server: What?
```

The symbol `>` is not part of the input, it just marks the input line.

§5. Serving multiple long-connected clients

In the case, when 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 only the modified client code 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(); // reading the next client message
    output.writeUTF(msg); // resend it to the client
}
```

It will work perfectly for interacting with a single client:

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

Received from 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 until responding to all messages from the first client. The reason is we use only a single thread for processing messages from all clients.

§6. Multithreaded server

The simplest way to work with multiple clients at the same time is to use multithreading! Let one server thread (e.g. `main`) accept new clients, while others interact with already accepted clients (1 thread for 1 client).

Here is our modified server with a new abstraction called `Session`. Since a session works in a separated thread, we can run multiple sessions at the same time.

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
    private static final int PORT = 34522;

    public static void main(String[] args) {
        try (ServerSocket server = new ServerSocket(PORT)) {

            while (true) {

                Session session = new Session(server.accept());

                session.start(); // it does not block this thread

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}


class Session extends Thread {

    private final Socket socket;


    public Session(Socket socketForClient) {

        this.socket = socketForClient;

    }


    public void run() {

        try (

    DataInputStream input = new DataInputStream(socket.getInputStream());


    DataOutputStream output = new DataOutputStream(socket.getOutputStream(
))

        ) {

            for (int i = 0; i < 5; i++) {

                String msg = input.readUTF();

                output.writeUTF(msg);

            }

            socket.close();

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

If we start this version of the server and several clients, all clients will be able to receive messages.

Important, that we do not use **try-with-resources** for `server.accept()` because we create a socket in one thread and close it in another thread. In this case, **try-with-resources** is very error-prone because one thread may close the socket while another thread wants to use it.

We hope you enjoy developing your own server in Java! Now you may use this knowledge to create a chat or a file storage.

Report a typo

106 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

This content was created about 2 years ago and updated 9 days ago. Share your feedback below in comments to help us improve it!

Comments (7) Hints (0) Useful links (1) Show discussion