UNIT 7
SOCKET FOR SERVERS

LH - 5HRS

PRESENTED BY: **ER. SHARAT MAHARJAN**

NETWORK PROGRAMMING

PRIME COLLEGE, NAYABAZAAR

CONTENTS (LH - 5HRS)

7.1 Using ServerSockets: Serving Binary Data, Multithreaded Servers, Writing to Servers with Sockets and Closing Server Sockets

7.2 Logging: What to Log and How to Log

7.3 Constructing Server Sockets: Constructing Without Binding

7.4 Getting Information about Server Socket

7.5 Socket Options: SO_TIMEOUT, SO-RSUMEADDR, SO_RCVBUF and Class of Service

7.6 HTTP Servers: A Single File Server, A Redirector and A Full-Fledged HTTP Server

7.1 Using ServerSockets

The **ServerSocket class contains everything needed to write servers** in Java. It has constructors that create new ServerSocket objects, methods that listen for connections on a specified port, methods that configure the various server socket options, and the usual miscellaneous methods such as toString( ) .

In Java, the basic life cycle of a server program is:

1. A new ServerSocket is created on a particular port using a ServerSocket( ) constructor.
2. The ServerSocket listens for incoming connection attempts on that port using its accept( ) method. accept( ) blocks until a client attempts to make a connection, at which point accept( ) returns a Socket object connecting the client and the server.
3. Depending on the type of server, either the Socket's getInputStream() method, getOutputStream( ) method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both close the connection.
6. The server returns to step 2 and waits for the next connection.

Example 1: A multithreaded daytime server

a. DaytimeServer.java

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DaytimeServer {
        public static void main(String[] args) {
                try {
                        ServerSocket server = new ServerSocket(1234);
                        while (true) {
                                try {
                                        Socket socket = server.accept();
                                        Thread task = new DaytimeThread(socket);
                                        task.start();
                                 } catch (IOException ex) {}
                        }
                 } catch (IOException ex) {
                        System.err.println("Couldn't start server");
                }}
private static class DaytimeThread extends Thread {
        private Socket socket;
        DaytimeThread(Socket socket) {
                this.socket = socket;
        }
        @Override
        public void run() {
                 try {
                        Writer out = new OutputStreamWriter(socket.getOutputStream());
                        Date now = new Date();
                        out.write(now.toString() +"\r\n");
                        out.flush();
                       socket.close();
                 } catch (IOException ex) {
                        System.err.println(ex);
                 }}}}
```

b. DaytimeClient.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

public class DaytimeClient {
    public static void main(String[] args) {
        String hostname = "localhost";
        int port = 1234;

        try (Socket socket = new Socket(hostname, port)) {
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));

            String serverTime = input.readLine();
            System.out.println("Server time: " + serverTime);
        } catch (IOException ex) {
            System.out.println("Client exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

Multithreaded Binary Server:

https://github.com/Sharatmaharjan/Np/blob/main/code/multithreadedBinaryServer.java

7.2 Logging

What to Log

1. **Server Start and Shutdown**: Log when the server starts and stops, including the port number it's listening on.
2. **Client Connections**: Log when a client connects to or disconnects from the server.
3. **Data Transmission**: Log significant data transmissions, such as requests and responses.
4. **Errors and Exceptions**: Log any exceptions or errors that occur, along with their stack traces.
5. **Thread Management**: Log the creation and termination of threads, especially in a multithreaded server.
6. **Requests and Responses**: Log the details of requests received and responses sent to clients.
7. **Configuration Changes**: Log changes to the server configuration, such as port numbers or security settings.
8. **Performance Metrics**: Optionally log performance metrics, such as the time taken to process requests.

How to Log

1. **Use a Logging Framework**: Use a robust logging framework like `java.util.logging`, Log4j, or SLF4J with Logback. These frameworks provide advanced features such as logging levels, appenders, and formatters.
2. **Define Logging Levels**: Use appropriate logging levels (e.g., `SEVERE`, `WARNING`, `INFO`, `DEBUG`, `TRACE`) to categorize the importance of log messages.
3. **Log Configuration**: Configure the logging framework to specify log file locations, log rotation policies, and log formats.
4. **Consistent Log Messages**: Ensure that log messages are consistent and provide enough context to understand what happened.

There are seven levels defined as named constants in java.util.logging.Level in

descending order of seriousness:

- Level.SEVERE (highest value-1000)
- Level.WARNING
- Level.INFO
- Level.CONFIG
- Level.FINE
- Level.FINER
- Level.FINEST (lowest value-300)

Logging Example

https://github.com/Sharatmaharjan/Np/blob/main/code/Logging.java

7.3 Constructing Server Sockets

There are **four public ServerSocket** constructors:

public ServerSocket(int port) throws BindException, IOException

public ServerSocket(int port, int queueLength) throws BindException, IOException

public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws IOException

public ServerSocket() throws IOException

For example, to create a server socket that would be used by an HTTP server on port 80, you would write:

ServerSocket httpd = new ServerSocket(80);

To create a server socket that would be used by an HTTP server on port 80 and queues up to 50 unaccepted connections at a time:

ServerSocket httpd = new ServerSocket(80, 50);

Example 2: Look for local ports.

```
import java.io.*;
import java.net.*;
public class LocalPortScanner {
	public static void main(String[] args) {
		for (int port = 1; port <= 65535; port++) {
			try {
				// the next line will fail and drop into the catch block if
				// there is already a server running on the port
				ServerSocket server = new ServerSocket(port);
			} catch (IOException ex) {
				System.out.println("There is a server on port " + port + ".");
			}
		}
	}
}
```

7.4 Getting Information about Server Socket

- The ServerSocket class provides two getter methods that tell you the local address and port occupied by the server socket.

public InetAddress getInetAddress()

- This method returns the address being used by the server (the local host).

public int getLocalPort()

- The ServerSocket constructors allow you to listen on an unspecified port by passing 0 for the port number. This method lets you find out what port you're listening on.

Example 3: A random port

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

public class RandomPort {
 public static void main(String[] args) {
 try {
 ServerSocket server = new ServerSocket(0);
 System.out.println("This server runs on port "+ server.getLocalPort());
 } catch (IOException ex) {
 System.err.println(ex);
 }}}
```

7.5 Socket Options

For server sockets, Java supports three options:

1. SO_TIMEOUT
2. SO_REUSEADDR
3. SO_RCVBUF

1. SO_TIMEOUT

- SO_TIMEOUT is the amount of time, in milliseconds, that accept() waits for an incoming connection. If SO_TIMEOUT is 0, accept() will never time out.
- The default is to never timeout.
- The setSoTimeout() method sets the SO_TIMEOUT field for the server socket object.
- The getSoTimeout() method returns the server socket's current SO_TIMEOUT value.

public void setSoTimeout(int timeout) throws SocketException

public int getSoTimeout() throws IOException

2. SO_REUSEADDR

- It determines whether a new socket will be allowed to bind to a previously used port while there might still be data traversing the network addressed to the old socket.
- There are two methods to get and set this option:

public boolean getReuseAddress() throws SocketException

public void setReuseAddress(boolean on) throws SocketException

- This code fragment determines the default value by creating a new ServerSocket and then calling getReuseAddress():

```
ServerSocket ss = new ServerSocket(10240);
System.out.println("Reusable: " + ss.getReuseAddress());
```

3. SO_RCVBUF

- The SO_RCVBUF option sets the default receive buffer size for client sockets accepted by the server socket.
- It's read and written by these two methods:

public int getReceiveBufferSize() throws SocketException

public void setReceiveBufferSize(int size) throws SocketException

- Setting SO_RCVBUF on a server socket is like calling setReceiveBufferSize().

Socket Options Programs

https://github.com/Sharatmaharjan/Np/blob/main/code/serverSocketOptions.java

- Different types of Internet services have different performance needs.
- For instance, live streaming video of sports needs relatively high bandwidth. On the other hand, a movie might still need high bandwidth but be able to tolerate more delay and latency. Email can be passed over low-bandwidth connection.
- Four general traffic classes are defined for TCP:

1. Low cost
2. High reliability
3. Maximum throughput
4. Minimum delay

7.6 HTTP Servers

- HTTP is a large protocol. A full-featured HTTP server must respond to requests for files, convert URLs into filenames on the local system, respond to POST and GET requests, handle requests for files that don’t exist, and much, much more.
- However, many HTTP servers don’t need all of these features.

1. Single File Server

A single file server serves a specific file in response to HTTP requests. This is useful for serving static content like an image or a specific HTML file.

2. Redirector

A redirector responds to requests with an HTTP redirection to another URL. This is useful for forwarding requests to a different server or URL.

3. Full-Fledged HTTP Server

A full-fledged HTTP server handles multiple types of requests, serves different types of content, and may include additional features like handling POST requests, serving dynamic content, etc.

https://github.com/Sharatmaharjan/Np/blob/main/code/HTTPservers.java

THANK YOU FOR YOUR ATTENTION

PREPARED BY : ER. SHARAT MAHARJAN