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

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class Example1 {
        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);
                }}}}
```

7.2 Logging

There are two primary things you want to store in your logs:

- Requests
- Server errors

Many legacy programs dating back to Java 1.3 and earlier still use third-party logging libraries such as log4j or Apache Commons Logging, but the **java.util.logging** package available since Java 1.4 suffices for most needs.

Choosing it avoids a lot of complex third-party dependencies. Although you can load a logger on demand, it’s usually easiest to just create one per class like so:

private final static Logger auditLogger = Logger.getLogger("requests");

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)

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 in- coming connection. If SO_TIMEOUT is 0, accept() will never time out.
- The default is to never time out.
- 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().

- 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.

THANK YOU FOR YOUR ATTENTION