UNIT-6

5 |

Unit 6: Socket for Clients

6.1 Introduction to Socket

6.2 Using Sockets: Investigating Protocols with telnet, Reading from Servers with Sockets, Writing to Servers with Sockets

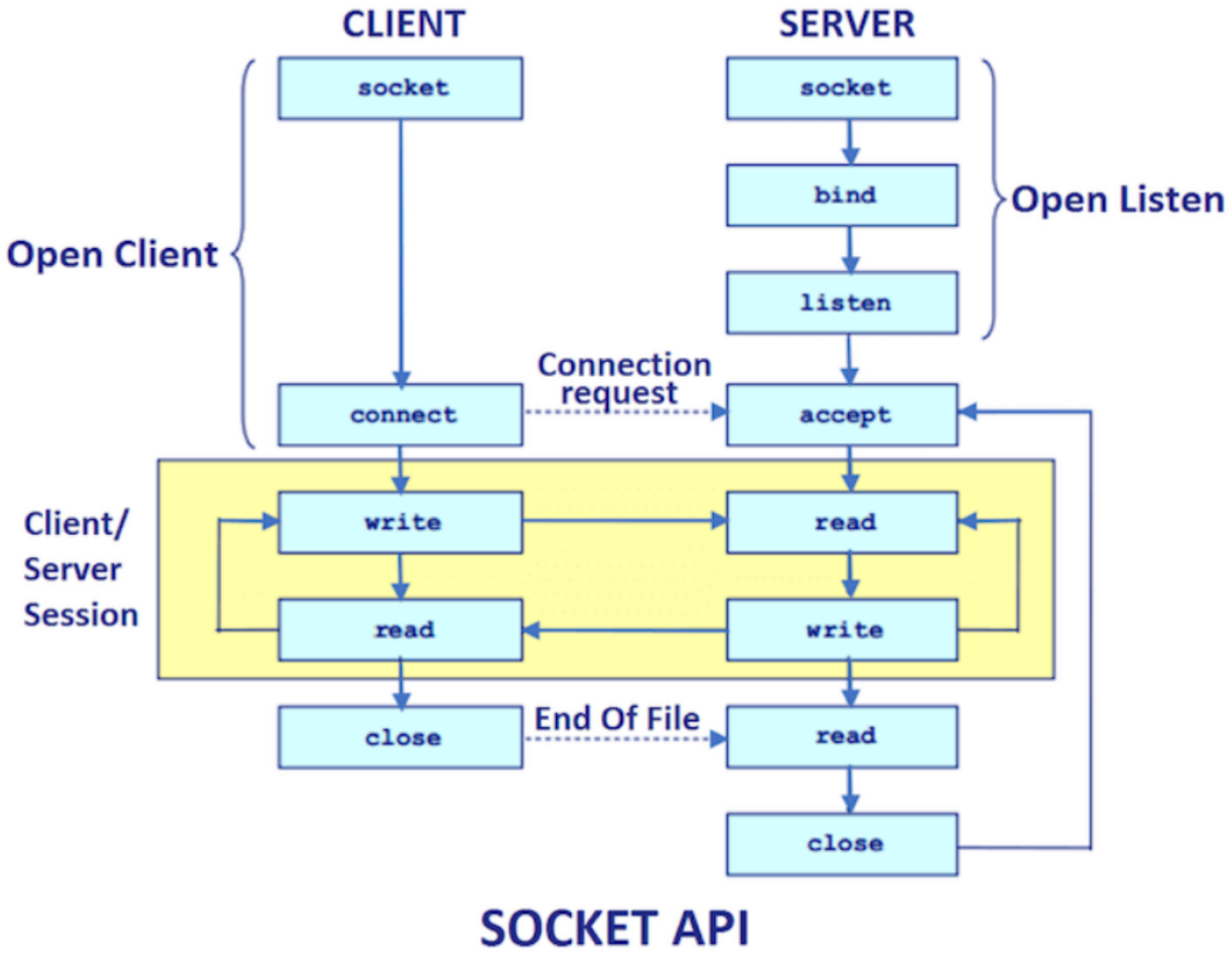
6.3 Constructing and connecting Sockets: Basic Constructors, Picking a Local Interface to Connect From, Constructing Without Connecting, Socket Addresses and Proxy Servers

6.4 Getting Information about a Socket: Closed or Connected?, toString()

6.5 Setting Socket Options: TCP_NODELAY, SO_LINGER, SO_TIMEOUT, SO_RCVBUF and SO_SNDBUF, SO_KEEPALIVE, OOBINLINE, SO_REUSEADDER and IP_TOS Class of Services

6.6 Socket in GUI Applications: Whois and A Network Client Library

SOCKETS


CLIENT
SERVER
socket
socket
bind
listen
Open Client
Open Listen
Connection request
connect
accept
Client/ Server Session
write
read
read
write
close
End Of File
read
close
SOCKET API

SOCKETS

- Sockets are a fundamental concept in network programming that **allow clients and servers to communicate with each other over a network**. A socket **is an endpoint** for **sending or receiving data between two entities on a network**.
- In order for a client to establish a connection with a server using sockets, the client needs to create a socket and specify **the IP address** and **port number of the server** it wants to connect to. The following steps can be taken to create a socket for a client:
- in network programming, a socket is an **endpoint that enables communication between two different processes over a network**. A socket is essentially a combination of an **IP address and a port number**, and it allows processes on different devices to exchange data.
- There are two types of sockets: **client sockets** and **server sockets**. **Client sockets initiate** communication, while **server sockets listen for incoming communication requests**.

The process of using sockets in a client involves the following steps:

1. Import the necessary packages for the client program. This can be done using the import statement at the top of your Java file.

```
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;
```

2. Creating a socket: The first step is to create a socket object using the Socket class, which takes two arguments: the IP address or hostname of the server, and the port number to connect to.

```
String serverAddress = "localhost";
int port = 8080;
Socket socket = new Socket(serverAddress, port);
```

3. Create DataInputStream and DataOutputStream objects to send and receive data to and from the server.

```
DataInputStream in = new DataInputStream(socket.getInputStream());
String message = in.readUTF();
System.out.println("Received message from server: " + message);
```

```
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
out.writeUTF(msg);
```

4. Close the input stream, output stream, and socket when you are done communicating with the server.

```
socket.close();
```

Client program read from server n write to server

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

public class Client {
    Run | Debug
    public static void main(String[] args) throws IOException {
        String serverAddress = "localhost";
        int port = 8080;
        Socket socket = new Socket(serverAddress, port);
        System.out.println("Connected to server at " + serverAddress + ":" + port);
        System.out.println(x:"enter a message: ");
        Scanner sc = new Scanner(System.in);
        String msg = sc.nextLine();
        sc.close();
        DataOutputStream out = new DataOutputStream(socket.getOutputStream());
        out.writeUTF(msg);
        DataInputStream in = new DataInputStream(socket.getInputStream());
        String message = in.readUTF();
        System.out.println("Received message from server: " + message);
        socket.close();
    }
}
```

The process of using sockets in a client involves the following steps:

1. Create a ServerSocket object that listens for incoming client connections on a specified port number. For example:

```
ServerSocket serverSocket = new ServerSocket(1234);
```

2. Use the **accept**() method of the ServerSocket class to wait for an incoming client connection. When a connection is accepted, the method returns a Socket object that represents the connection to the client. For example:

```
Socket clientSocket = serverSocket.accept();
```

3.Create an ObjectInputStream and an ObjectOutputStream object to read and write data to the client socket using the getInputStream() and getOutputStream() methods of the Socket class. For example:

```
ObjectInputStream inputStream = new ObjectInputStream(clientSocket.getInputStream());
ObjectOutputStream outputStream = new ObjectOutputStream(clientSocket.getOutputStream());
```

4. Use the **writeUTF()** method of the ObjectOutputStream object to send data to the client. For example:

```
outputStream.writeUTF("Hello, client!");
outputStream.flush();
```

The process of using sockets in a client involves the following steps:

5.Use the readUTF() method of the ObjectInputStream object to receive data from the client. For example:

```
String message = inputStream.readUTF();
```

6.Close the input and output streams and the client socket when you are finished communicating with the client. For example:

```
clientSocket.close();
```

Server program Read From client and write to server

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

public class Server {
    Run | Debug
    public static void main(String[] args) throws IOException {
        int port = 8080;
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("Server started on port " + port);
            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected: " + clientSocket.getInetAddress().getHostAddress());
                DataInputStream in = new DataInputStream(clientSocket.getInputStream());
                String message = in.readUTF();
                System.out.println("Received message from client: " + message);
                DataOutputStream out = new DataOutputStream(clientSocket.getOutputStream());
                out.writeUTF("Server received message: " + message);
                clientSocket.close();
            }
        }
    }
}
```

Investigating Protocols with Telnet

To investigate a protocol using Telnet, you can use Telnet to connect to a server that implements the protocol and interact with the **server by sending commands and receiving responses**.

Here are the general steps to follow:

1. Determine the **host and port number** of the server that implements the protocol you want to investigate.
2. **Open a command prompt** or terminal window and type the following command to **start Telnet**:
3. **telnet hostname port (o www.google.com 80) o for open**
4. Replace hostname with the hostname or IP address of the server, and replace port with the port number that the server listens on.
5. Once you are connected, you can start sending commands to the server. The commands will depend on the protocol you are investigating. For example, if you are investigating the HTTP protocol, you can send an HTTP request to the server, such as:

GET / HTTP/1.1

Host: google.com

1. This will request the root page of the **google.com website**.
2. After sending the command, press Enter to send it to the server. The server will then send a response, which you can read in the Telnet window.
3. Continue sending commands and reading responses to investigate the protocol
4. When you are finished, you can type **quit or exit** to terminate the Telnet session and close the connection.

Investigating Protocols with Telnet

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

public class DaytimeServer {
    Run | Debug
    public static void main(String[] args) {
        // Get the port number from the command line argument
        int port = Integer.parseInt(args[0]);
        try {
            // Create a new server socket and bind it to the specified port
            ServerSocket serverSocket = new ServerSocket(port);
            while (true) {
                // Wait for a client to connect
                Socket clientSocket = serverSocket.accept();
                // Get the current date and time
                String date = new Date().toString();
                // Send the date and time to the client using writeUTF
                DataOutputStream outToClient = new DataOutputStream(clientSocket.getOutputStream());
                outToClient.writeUTF("Today is :" + date);
                // Close the connection
                clientSocket.close();
            }
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
```

CONSTRUCTING AND CONNECTING SOCKETS

- In Java, the Socket class has two commonly used constructors that create a client-side socket object:
- public Socket(String host, int port) throws UnknownHostException, IOException:
- This constructor creates a new Socket object that connects to the server at the specified **host name or IP address and port number**. It throws an UnknownHostException if the specified host cannot be resolved to an IP address, and an IOException if there is an error while connecting to the server.

public Socket(InetAddress host, int port) throws IOException:

This constructor creates **a new Socket** object that connects to the server at the specified **InetAddress object and port number**. An InetAddress object represents an **IP address**, and can be obtained using the InetAddress.getByName() method. This constructor throws an IOException if there is an error while connecting to the server.

CONSTRUCTING WITHOUT CONNECTING

- In Java, you can create a Socket object **without establishing a connection to a remote host**. This is useful when you want to listen for incoming connections or when you want to configure the socket before connecting.
- To create a Socket object without connecting, you can use one of the following constructors:
- **public Socket()**
- **public Socket(String host, int port)**
- **public Socket(InetAddress address, int port)**
- The first constructor creates a new Socket object **without specifying a remote host or port**. You can use this constructor when you want to create a **socket that will listen for incoming connections**.
- The second constructor creates a new Socket object and **specifies the remote host and port** that you want to connect to. However, it actually establish the connection. You can use this constructor to **set up the socket configuration before connecting**.
- The third constructor is similar to the second constructor, but instead of taking a host name as a parameter, it takes an **InetAddress object** representing the **remote host's IP address**.

CONSTRUCTING WITHOUT CONNECTING

```
import java.net.*;

public class SocketExample {

    public static void main(String[] args) throws Exception {

        Socket socket = new Socket();

        socket.setReuseAddress(true);

        socket.setSoTimeout(5000);

        socket.bind(new InetSocketAddress("localhost", 1234));

    }

}
```

PORT SCANNING PROGRAM

```java
import java.net.InetSocketAddress;
import java.net.Socket;

public class PortScanner {
    Run | Debug
    public static void main(String[] args) {
        String host = "localhost"; // Replace with the host you want to scan

        for (int port = 1; port <= 2048; port++) {
            try {
                Socket socket = new Socket();
                socket.connect(new InetSocketAddress(host, port), timeout:1000);
                System.out.println("Port " + port + " is open");
                socket.close();
            } catch (Exception e) {
                // Port is closed or host is unreachable
            }
        }
    }
}
```

SOCKET ADDRESSES

The Socket class in Java provides two methods to retrieve the **local and remote addresses** associated with the socket connection: **getLocalSocketAddress()** and **getRemoteSocketAddress().**

Here's a brief explanation of each method:

1. **getLocalSocketAddress**(): Returns the local address of the socket connection as a SocketAddress object. This address includes the **IP address and port number of the local host**.

```
SocketAddress localAddress = socket.getLocalSocketAddress();
```

2. getRemoteSocketAddress(): Returns the **remote address** of the socket connection as a SocketAddress object. This address includes **the IP address and port number** of the remote host that the socket is connected to.

```
SocketAddress remoteAddress = socket.getRemoteSocketAddress();
```

Both of these methods return a SocketAddress object, which is an **abstract class that represents a socket address**. You can cast this object to a InetSocketAddress or SocketAddress depending on the type of socket address you want to work with.

SOCKET ADDRESSES

```java
import java.net.*;
public class LocalRemote {
    Run | Debug
    public static void main(String[] args) throws Exception {
        String hostname = "www.google.com";
        int port = 80;
        try (// Create a socket and connect to the remote server
                Socket socket = new Socket(hostname, port)) {
            // Retrieve the local and remote socket addresses
            InetSocketAddress localAddr = (InetSocketAddress) socket.getLocalSocketAddress();
            InetSocketAddress remoteAddr = (InetSocketAddress) socket.getRemoteSocketAddress();

            // Print the local and remote socket addresses
            System.out.println("Local address: " + localAddr.getAddress().getHostAddress() + ":" + localAddr.getPort());
            System.out.println(
                    "Remote address: " + remoteAddr.getAddress().getHostAddress() + ":" + remoteAddr.getPort());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
Local address: 192.168.1.70:52985
Remote address: 142.250.193.132:80
```

PROXY SERVERS

The **public Socket(Proxy proxy) constructor** in Java creates **a new instance of the Socket class** using the **specified Proxy object**.

A Proxy object represents a proxy server that acts as an intermediary between the client (the Socket object) and the server being accessed. By using a proxy server, the **client can make requests to the server without revealing its own IP address, location, or other identifying information.**

```
// Create a new proxy object for an HTTP proxy server at 192.168.0.1:8080
Proxy proxy = new Proxy(Proxy.Type.HTTP,
    new InetSocketAddress(hostname:"192.168.0.1", port:8080));

// Create a new Socket object using the proxy
Socket socket = new Socket(proxy);

// Use the Socket object to connect to a server
socket.connect(new InetSocketAddress("www.example.com", 80));
```

GETTING INFORMATION ABOUT A SOCKET

In Java, we can get information about a Socket object using the following methods:

- getInetAddress() - **returns the remote address** (i.e., IP address) of the socket as an InetAddress object.
- getPort() - returns the **remote port number to which the socket is connected**.
- getLocalAddress() - returns the **local address to which the socket is bound** as an InetAddress object.
- getLocalPort() - returns the **local port number to which the socket is bound**.

```java
import java.net.*;
public class SocketInfo {
    Run | Debug
    public static void main(String[] args) throws Exception {
        // Create a new Socket object and connect it to a server
        Socket socket = new Socket(host:"www.google.com", port:80);

        // Get information about the socket
        InetAddress remoteAddr = socket.getInetAddress();
        int remotePort = socket.getPort();
        InetAddress localAddr = socket.getLocalAddress();
        int localPort = socket.getLocalPort();

        // Print the socket information
        System.out.println("Remote address: " + remoteAddr);
        System.out.println("Remote port: " + remotePort);
        System.out.println("Local address: " + localAddr);
        System.out.println("Local port: " + localPort);

        // Close the socket
        socket.close();
    }
}
```

CLOSED OR CONNECTED? AND Tostring()

In Java, you can check whether a Socket object is closed or connected using the following methods:

isClosed() - returns true if the **socket has been closed**, false otherwise.

isConnected() - returns true if the **socket is connected to a remote host**, false otherwise.

The toString() method in the Socket class returns a **string representation of the socket**, which includes the **remote IP address and port number, as well as the local IP address and port number**.

```
Socket socket = new Socket("www.google.com", 80);

    // Get a string representation of the socket

    String socketStr = socket.toString();

    // Print the string representation

    System.out.println(socketStr);
```

```
Socket[addr=www.google.com/93.184.216.34,port=80,localport=52825]
```

SOCKET OPTION

TCP_NODELAY: This option disables the **Nagle algorithm**, which combines small outgoing messages into a larger packet **to reduce network overhead**. Setting this option can **improve performance for applications that send many small messages**.

SO_BINDADDR: This option sets the **local address the socket should bind to**. It can be useful when a system has **multiple network interfaces and you want to specify which interface to use.**

SO_TIMEOUT: This option sets the timeout for blocking socket operations, such as read() and write(). If the **timeout expires** before the operation completes, a SocketTimeoutException is thrown.

SO_LINGER: This option controls what happens when a socket is closed and there is unsent data in the send buffer. If the SO_LINGER option is **set to a non-zero timeout value,** the close() method will block until either all data has been sent, or the timeout expires. If the timeout expires before all data is sent, the socket is closed with an error.

SO_SNDBUF: This option **sets the size of the socket's send buffer**, which is used to hold outgoing data before it is sent over the network.

SO_RCVBUF: This option **sets the size of the socket's receive buffer**, which is used to hold incoming data before it is read by the application.

SO_KEEPALIVE: This option enables or disables **the TCP keep-alive mechanism**, which sends periodic packets to check if the connection is still alive.

OOBINLINE: This option **enables or disables the ability to send and receive out-of-band (OOB) data**, which is data that has a higher priority than normal data.

IP_TOS: This option sets the **Type of Service** (ToS) field in **the IP header**, which is used to **prioritize network traffic** based on the desired level of service.

SOCKET OPTION

```
// Create a new Socket object and connect it to a server
Socket socket = new Socket(host:"www.google.com", port:80);
// Disable the Nagle algorithm to improve performance
socket.setOption(SocketOption.TCP_NODELAY, value:true);
// Bind the socket to a specific local address
InetAddress localAddress = InetAddress.getByName(host:"192.168.1.100");
socket.setOption(SocketOption.SO_BINDADDR, localAddress);
// Set a timeout of 10 seconds for blocking operations
socket.setOption(SocketOption.SO_TIMEOUT, value:10000);
// Set a linger timeout of 5 seconds
socket.setOption(SocketOption.SO_LINGER, value:5);
// Increase the send buffer size to 64 KB
socket.setOption(SocketOption.SO_SNDBUF, 64 * 1024);
// Increase the receive buffer size to 128 KB
socket.setOption(SocketOption.SO_RCVBUF, 128 * 1024);
// Enable TCP keep-alive
socket.setOption(SocketOption.SO_KEEPALIVE, value:true);
// Enable the ability to send and receive out-of-band data
socket.setOption(SocketOption.OOBINLINE, value:true);
// Set the Type of Service field to high priority
socket.setOption(SocketOption.IP_TOS, value:0x10);
```

BASIC STEPS GUI APPLICATIONS: whois

- The basic steps involved in the working process of a Whois application using sockets in a GUI application are:

1. The user enters a domain name in the input field of the GUI.
2. When the user clicks the "Lookup" button, the application creates a socket and connects to the Whois server using the socket.
3. The application sends a query containing the domain name to the server using the socket's output stream.
4. The server processes the query and sends a response containing information about the domain name back to the client using the socket's input stream.
5. The application receives the response from the server using the socket's input stream.
6. The application parses the response to extract the information about the domain name.
7. The application displays the information about the domain name in the output field of the GUI.

WHAT IS SOCKETS IN GUI APPLICATIONS: whois

- Sockets in **GUI applications** are a way to connect to and communicate with **servers over the network**. In the context of a Whois application, **a socket is used to connect to a Whois server** and **retrieve information about a domain name**.
- A Whois server is a **database of information about domain names and the entities** that own or administer them. When you enter a domain name into a Whois application, the application **uses a socket to connect to the Whois server**, sends a query containing the domain name, and **receives a response containing information about the domain name**.

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
public class WhoisClient extends JFrame implements ActionListener {
    private JTextField domainField;
    private JTextArea resultArea;
    public WhoisClient() {
        super(title:"Whois Client");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        domainField = new JTextField();
        add(domainField, BorderLayout.NORTH);
        resultArea = new JTextArea();
        add(new JScrollPane(resultArea), BorderLayout.CENTER);
        JButton lookupButton = new JButton(text:"Lookup");
        lookupButton.addActionListener(this);
        add(lookupButton, BorderLayout.SOUTH);
        setSize(width:400, height:300);
        setVisible(b:true);
    }
    public void actionPerformed(ActionEvent e) {
        String domain = domainField.getText();
        String result = lookup(domain);
        resultArea.setText(result);
    }
    private String lookup(String domain) {
        try {
            Socket socket = new Socket(host:"whois.internic.net", port:43);
            DataOutputStream out = new DataOutputStream(socket.getOutputStream());
            DataInputStream in = new DataInputStream(socket.getInputStream());
            out.writeUTF(domain);
            String response = in.readUTF();
            socket.close();
            return response;
        } catch (IOException e) {
            return "Error: " + e.getMessage();
        }
    }
    Run | Debug
    public static void main(String[] args) {
        new WhoisClient();
    }
}
```